Device Drivers: a Comparison of Different Development Strategies

J.J. Loubser

Thesis presented in partial fulfilment of the requirements for the degree of
Master of Science
at the University of Stellenbosch

Supervisor: P.J.A. de Villiers

March 2000

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Abstract

Users are not supposed to modify an operating system kernel, but it is often necessary to add a device driver for a new peripheral device. Device driver development is a difficult and time-consuming process that must be performed by an expert. Drivers are difficult to debug and a malfunctioning driver could cause the operating system to crash. Ways are therefore needed to make the development of device drivers safer and easier.

A number of different device driver development methods are examined in this thesis. An existing micro-kernel that supports in-kernel device drivers as well as extensible device drivers has been modified to support user-level and loadable drivers. These extensions ensured that all the development methods were implemented in the same environment and a comparison could thus be made on a fair basis.

A comparison of the different methods with respect to the efficiency of the resulting device driver, as well as the ease of the development process, is presented.

Opsomming

Gebruikers is nie veronderstel om aan 'n bedryfstelsel te verander nie, maar tog is dit gereeld nodig om 'n toesteldrywer vir 'n nuwe randapparaat by te voeg. Die ontwikkeling van 'n toesteldrywer is 'n tydrowende en moeilike proses en moet deur 'n kundige aangepak word. Toesteldrywers is moeilik om te ontfout en kan deur verkeerde werking die hele stelsel tot stilstand bring. Daar is dus tegnieke nodig om die ontwikkeling van toestelhanteerders makliker en veiliger te maak.

'n Aantal verskillende ontwikkelingsmetodes vir toesteldrywers word in hierdie tesis ondersoek. 'n Bestaande mikro-kern wat in-kern, sowel as uitbreibare toesteldrywers ondersteun, is aangepas om gebruikersvlak en laaibare toestelhanteerders te ondersteun. Hierdie uitbreiding het verseker dat al die ontwikkelingsmetodes in dieselfde omgewing geïmplementeer is. Dit was dus moontlik om die metodes op 'n regverdige grondslag te vergelyk. Die vergelyking is gedoen ten opsigte van die effektiwiteit van die resulterende toesteldrywer sowel as die moeilikheidsgraad van die ontwikkelingsproses.

Acknowledgements

I am grateful to the following for their invaluable assistance throughout my studies:

- my wife, Gayle, for her help, encouragement and support, without which this thesis might never have been completed;
- Pieter de Villiers for guidance and advice;
- Pieter Muller who was always ready to answer my many questions;
- fellow members of the Hybrid team;
- the Foundation for Research Development (FRD) and the University of Stellenbosch for financial support.

Dedicated to Gayle for her love
and to my parents, for giving me a love of learning

Contents

Tables

Figures

Chapter 1

Introduction

An operating system is a complex and vital part of most computer systems. It is important to design and implement the operating system correctly, since it is the base upon which the rest of the system is built. Once an operating system has been stabilized, the most common modification is the addition of new device drivers. The development of device drivers must be undertaken by an expert, because it is a difficult and time consuming process. This thesis compares different techniques to make the development of device drivers safer and easier.

A device driver should hide the complex hardware details of the device by providing a simple interface to the rest of the system. This interface often hides the asynchronous nature of input/output (I/O) operations on peripheral devices. The abstraction provided by a device driver is a successful way of making application programming easier. However, the implementation of the device driver itself remains a painstaking, error prone process.

A device driver interacts on a low level with hardware, as well as with an operating system kernel. This means that a particular implementation of a driver is specific to the hardware and the kernel. Therefore a good knowledge of the internal details of both the operating system kernel and the peripheral device is required. This is not an easy requirement to meet, since operating system kernels are usually complex and difficult to understand. Another problem is that documentation for peripheral devices is often incomplete, incorrect or misleading.

The design of a device driver is influenced by the structure of the peripheral device. For example, an inexpensive device such as a floppy drive has a simple device controller and much of the low level work to control the hardware device must be performed by the device driver.

The speed of the device in relation to the processor is often a critical factor in implementing a device driver. The device might produce more data than the processor can handle, for example a high speed network card can send or receive megabytes of data per second. To solve this problem, elaborate workarounds must be implemented to operate the device at optimal speed without losing data. The opposite is also sometimes true: the device can be slower than the processor, in which case the driver must wait until the peripheral device has completed an operation. Because of the speed difference, some errors appear only when a certain sequence of events occurs, influenced by the system load and the speed of the device. These errors can be very difficult to find, since debugging and inspection aids like debuggers and trace output alter the sequence of events that lead to the error and could make it disappear.

Various strategies to develop device drivers have been implemented as part of operating systems, each with different advantages and disadvantages. In this thesis, a number of strategies are compared with special emphasis on how easy it is to implement drivers and how the strategies affect the efficiency and safety of the device driver. The strategies are:

- **In-kernel device drivers:** These drivers are statically linked to the rest of the kernel, and have access to the full instruction set of the processor.
- **Device abstraction:** The aim of this strategy is to separate the kernel and the device driver by using a well-defined interface. A further development of this strategy is to provide an object oriented framework for the implementation of extensible drivers.
- **Loadable drivers:** The device driver code is loaded into the kernel at run-time. After being loaded, the driver executes as an in-kernel driver. This strategy improves the flexibility of the system and makes it easier to configure.
- **User-level drivers:** The device drivers execute as user-level processes. This strategy allows all the tools for the debugging and development of normal user-level processes to be used for debugging device drivers. It makes the system safer and easier to configure, but it is less efficient.

These strategies have been selected because they are easy to implement on the selected test-bed system and also because they are commonly used to implement device drivers on existing kernels. For example, *in-kernel device drivers* have been implemented in UNIX [8] and Amoeba [38, 40], *device abstraction* in Minix [39] and Mach[17], *loadable drivers* in Linux [3] and SPIN [6], and *user-level drivers* in L4 [28] and QNX [20]. However, strategies that are implemented in different kernels cannot be compared with any degree of accuracy, since

the underlying principles and mechanisms of the various kernels differ. An experiment was needed to implement the strategies in the same environment so that a fair comparison could be made.

The Gneiss micro-kernel was selected as the environment for this experiment, because it is representative of typical micro-kernels and is therefore suitable for making conclusions that are applicable to other micro-kernels [16, 17]. The kernel was developed at the University of Stellenbosch and is well-known to the author. It is written in a high-level language instead of assembler, which simplifies the experiment. Some of the necessary code, namely *in-kernel drivers* and *extensible drivers* had already been implemented prior to the start of this project. Therefore only *loadable drivers* and *user-level drivers* needed to be implemented as part of this thesis. Afterwards, it was possible to compare the development of device drivers using these strategies in a homogeneous environment.

The goal of this thesis is to study and compare the device driver development strategies outlined above. The goal of each investigation is twofold: to show how easy it is to develop and to maintain a device driver, and to show how efficient the driver is. Each strategy is evaluated with regard to these criteria. This evaluation is used to recommend the best strategy according to the design needs of the kernel developer.

The rest of this thesis has the following structure: Chapter 2 presents background information about the structure and development of device drivers, as well as an overview of the Gneiss kernel. Various strategies for the development of device drivers are also discussed, and a survey of how they are implemented in a number of well-known micro-kernels is given. The Gneiss micro-kernel is then examined to provide an understanding of the issues that are involved in the implementation of device drivers. Methods to develop user-level device drivers are discussed in Chapter 3. Two different programming models, thread-based and event-based, are examined and compared. The ways in which the design of the Gneiss kernel had to be modified to support these models are then discussed, after which a description is given of how these models were implemented on an Intel 80x86 processor. In Chapter 4 the various development strategies are evaluated. Chapter 5 concludes this thesis and provides an overview of the results, as well as a brief discussion of future work.

Chapter 2

Background

In the previous chapter it was stated that the goal of this thesis is to compare four different development strategies for device drivers. To allow for a fair comparison, these strategies must all be implemented in the same environment. The Gneiss kernel, which is a successor of the HYBRID kernel, has been selected as test-bed for this experiment [16, 17]. This chapter provides an overview of the internal detail of the kernel. The properties of device drivers will also be examined to provide an understanding of the various implementation issues that are not dependent on a specific operating system. The design of a device driver is influenced by the structure of the operating system kernel. Therefore the kernel support for each of the different development strategies and the influence that the kernel has on the implementation of device drivers will be discussed. This discussion will be illustrated with some examples of how device drivers are implemented in a number of related kernels.

2.1 Properties of a Device Driver

Peripheral devices differ widely in terms of their functionality and behaviour. This makes it difficult to develop a generic model to handle all peripheral devices and the operations that can be performed on them. In the UNIX operating system and its variants, devices are classified as either *character devices* or *block devices* [3, 39, 40]. A character device sends or receives a stream of data as a sequence of bytes that are not ordered into packets, for example a serial line. A block device works on the principle that data is stored in a number of blocks of a fixed size. The crucial property of a block device is that each block can be accessed independently from the other blocks. An example of a block device is a hard disk.

However, there are some devices that do not fit into this classification. Clocks and memory mapped devices are neither block nor character devices, while a tape drive can be classified as a character device or as a block device, depending on the viewpoint of the user.

Some kernels support Input/Output (I/O) systems that provide a common interface to all peripheral devices [1, 8, 15, 39]. This means that the same interface is used to access each device driver, which makes it easier to implement device independent software. For example, in UNIX each device is represented by a special file and a device is accessed by performing standard file operations like open, read or write. However, these Input/Output systems do not make it any easier to develop the underlying device drivers.

Most peripheral devices can accept a request, process it and signal to the processor when the state of the device has changed. This leaves the processor free to carry on with other processing while the device is busy. However, some simple hardware controllers do not have signalling capabilities. This means that the device driver must use polling to detect a change in the state of the peripheral device. This type of driver is often very simple, since it is not necessary to manage the additional complexity of having to field interrupts. The driver is also efficient, because it can respond immediately when the state of the device changes. However, the major disadvantage of this approach is that the processor wastes computing cycles while it spins in a tight loop. Although this is not a problem for single process systems since the processor is dedicated to one process anyway, it is unacceptable for an operating system that supports concurrency, because polling drivers monopolize the processor and prevent other processes from being executed. Therefore if a driver needs to wait for a peripheral device, the driver must block by yielding control of the processor and allowing the rest of the system to continue to execute. The driver will be resumed when the peripheral device signals that a change in its state has occurred, for example due to an error or because the current operation has been completed. This mechanism incurs overhead due to blocking and resuming the driver, but it leaves the processor free to perform other processing while the peripheral device is busy. Since most modern operating systems support concurrency, drivers typically make use of polling only when it is required by the peripheral hardware and then only for short intervals.

Some devices, like disk drives, are *shareable*, because client requests can be broken down into sub-requests that can be processed in any order. This is in contrast to *dedicated* devices, like serial lines, that must execute a client request to completion before the next request can be accepted. It is easier to write drivers for dedicated devices than to write drivers for shareable devices, because each request to a dedicated device is processed to completion before the

next request can be accepted. Therefore it is not necessary to manage a number of partially completed requests. However, in certain circumstances a device driver that can handle more than one request at a time is more efficient. For example, a driver for a hard disk that schedules incoming requests according the position of the disk head will improve throughput since the disk seek times are shortened, although the response time may suffer, since requests are not necessarily performed in first-in-first-out (FIFO) order.

A non-polling, shareable device driver does not need to complete a particular device request before a new request can be accepted and must also respond to events that arrive in an unpredictable order. Therefore it is an asynchronous server. Such a driver is the general case, but it can also be used to support special cases such as polling or non-shareable devices. The rest of this thesis will concentrate on development strategies for shareable, non-polling device drivers, since most device drivers are of this type and polling or non-shareable drivers can be developed using the same techniques.

The above issues show how the device hardware affect the complexity of the driver. For example, a polling driver is simpler, and therefore easier to implement, than a shareable, non-polling driver. However, there are other factors that also have an important impact on how easy it is to develop a device driver.

- **Hardware documentation:** The documentation for hardware devices is sometimes incomplete or ambiguous. This could cause the initial development phase to be time consuming, since the developer must determine the correct parameters for device operations by trial and error.
- **Kernel structure:** The complexity of the interaction between the device driver and the operating system kernel also influences how easy it is to implement the device driver. If the developer needs to have a detailed knowledge of the operating system to implement a driver, it makes the development process more time consuming and error prone.
- **Debugging support:** A suitable debugging environment and good debugging tools could improve the productivity of the device driver developer by allowing errors to be detected and corrected quickly.

An in depth discussion of *Hardware documentation* is beyond the scope of this thesis, but the influence of the *Kernel structure* and *Debugging support* on device driver development is relevant to determine the best strategy for developing device drivers.

2.2 Device Driver Development Strategies

The structure of an operating system has a great influence on the structure of the device drivers that are developed for that system. For example, in a monolithic operating system it is natural that device drivers are implemented within the kernel where they have direct access to the peripheral hardware. However, in a micro-kernel based operating system, the driver could conceivably execute in user space, like a normal program. In the following sections a number of different strategies to develop device drivers will be examined.

2.2.1 In-kernel Device Drivers

In a traditional monolithic operating system, device drivers form part of the kernel and users can access them only via system calls. It is a difficult and time-consuming process to implement and debug an in-kernel driver, since a thorough knowledge of the interaction between the driver and the kernel is needed. To debug such a driver, special software, for example a kernel that is compiled with debugging information, is often required. It is often necessary to reboot the kernel for each test run of the driver, which makes development time-consuming and frustrating. In-kernel drivers are also potentially dangerous, because a driver has access to the address space of the kernel, as well as to the complete instruction set of the processor. This means that a malfunctioning device driver could corrupt the memory of the operating system and bring the whole system to a halt. On the other hand, the greatest advantages of in-kernel drivers are efficiency and direct access to kernel resources.

2.2.2 Device Abstraction

In an attempt to bring greater structure and modularity to operating system design, many kernels are organized as a number of layers that are constructed on top of each other. This allows device drivers to be isolated from the rest of the kernel and makes it unnecessary for the developer to understand the rest of the kernel in detail, because all communication between the driver and the kernel is performed through a common interface. However, the driver is still statically linked to the kernel, meaning that the other disadvantages of Section 2.2.1 still apply: the driver is difficult to debug, could corrupt the system and the system must be rebooted for each test run.

In Minix, device drivers are written as separate processes that execute within the kernel [39].

```
message mess;                    /* message buffer */

void io_process() {
   initialize();                 /* only done once during system init */
   while (TRUE) {
      receive(ANY, &mess);       /* wait for a request for work */
      caller = mess.source;      /* process from whom message came */
      switch(mess.type) {
         case READ:   rcode = dev_read(&mess); break;
         case WRITE:  rcode = dev_write(&mess); break;
         /* Other cases go here, including OPEN, CLOSE and IOCTL */
         default:     rcode = ERROR;
      }
      mess.type = TASK_REPLY;
      mess.status = rcode;       /* result code */
      send(caller, &mess);       /* send reply message back to caller */
   }
}
```

Figure 1: Outline of a device driver in Minix

These drivers are fully functional processes with their own registers and stack. Driver processes are scheduled like normal processes and communicate with the rest of the kernel via interprocess communication. Figure 1 shows the outline of the main procedure of a typical device process in Minix. A device process uses the *receive* primitive to accept a request for work. After a request has been received, the device process attempts to fulfill the request and sends a reply message by executing the *send* primitive. The request must be completed before the next device request message can be accepted. Therefore a device process can only handle one request at a time. After a device process has accepted a request, it specifies that it wants to accept only interrupt messages, until the request had been completed. Interrupts are converted into messages and passed to the appropriate device driver process. While the device driver process is processing a device request, any messages other than interrupt messages are queued.

This approach abstracts a device driver from the kernel by treating it as a process that can be scheduled. The interface to the kernel is well defined and it extends easily to distributed systems. Due to this abstraction, device drivers are less efficient, primarily because of the scheduling of the device driver processes and the interprocess communication between the drivers and the kernel. However, the device drivers still have complete access to the hardware and the rest of the kernel.

Another form of abstraction is to develop generic device drivers that can be extended easily to

support new peripheral devices. This is done by first identifying classes of similar devices, for example various types of network cards. A device driver for such a family of devices is then divided into two parts: 1) a generic module that manages the flow of control of the driver and interacts directly with the kernel, and 2) a specific module, that provides device specific functionality and interacts only with the generic module. The generic module acts as a service module and provides abstract data structures and procedures that are called by the specific module. However, the generic module must be able to request the specific module to perform any device specific operations, therefore in that case it is necessary to invert the relationship and allow the generic module to call procedures within the specific module. This mechanism is called an *up-call* and its big advantage is that the generic module can initiate operations instead of always having to act as a passive library. This approach allows extensibility as well as code sharing, and isolates the developer of the specific module from the control flow and complexities of the device and the kernel.

```
Controller* = POINTER TO ControllerDesc;
Params* = POINTER TO ParamsDesc;

Start* = PROCEDURE(c: Controller; params: Params);
Transfer* = PROCEDURE(c: Controller; params: Params);
Cleanup* = PROCEDURE(c: Controller; params: Params);

ControllerDesc* = RECORD (Object) (* ControllerDesc is derived from Object *)
   waiting: Queue;       (* processes that are waiting for service *)
   activeproc: Process; (* process that is being serviced *)
   start: Start;         (* generic procedure for starting a device request *)
   transfer: Transfer;  (* generic procedure for transferring data *)
   cleanup: Cleanup;     (* generic procedure to perform cleanup *)
END;

ParamsDesc* = RECORD    (* record that defines the parameters of an operation *)
   controller: Controller;
   drive*, sector*, num*, bufadr*: LONGINT;
   operation*: SHORTINT;
END;
```

Figure 2: Generic module interface definition

The Gneiss micro-kernel that is implemented in the Oberon language has support for the development of extensible device drivers [32]. Oberon is the successor of Modula-2 and is a type safe, object oriented language [31, 34]. Object orientation is achieved through the use of type extension and procedure variables. In Oberon, objects are implemented with the *record* data structure. A record can contain variables that describe the state of an object, and procedure variables that encapsulate the operations that can be performed on that object. A

record can be derived from a parent record to inherit the functionality of the parent record. This operation is called type extension, since the base record is extended by the derived record to implement extra functionality.

Figure 2 shows the interface for an object that represents a generic disk controller. The code has been simplified and only the relevant details are shown. The controller object contains variables *waiting* and *activeproc* that contains respectively a queue of waiting processes and the process of which the request is currently being processed. The controller object also contains the procedure variables *start, transfer* and *cleanup* that represent the actions that can be performed on the device controller. The *ParamsDesc* object contains variables that describe the parameters of a specific request. This object is passed to the specific module as one of the parameters of the procedure variables in the Controller object.

```
PROCEDURE TransactionHandler(c: Object; request, reply: Message) : LONGINT;
VAR
   result: LONGINT;
   currentProc: Process;
   params: Params;
BEGIN
   WITH c: Controller DO                          (* cast c to type Controller *)
      CASE request.operation OF
         Read:
            currentProc := PreemptProcess(NIL);    (* preempt current process *)
            params := Parameters(request);         (* obtain device request parameters *)
            IF c.activeproc = NIL THEN
               c.activeproc := currentProc;
               c.start(c, params)                 (* up-call to start procedure *)
            ELSE
               Objects.Put(c.waiting, currentproc) (* queue device request *)
            END
        |Write: ...
      END
   END
END TransactionHandler;
```

Figure 3: Generic Transaction Handler

Figure 3 shows the procedure in the generic module that handles device requests from client processes. The *TransactionHandler* procedure is called when a client process requests an I/O operation. The user process is preempted and if the device is not busy, a new I/O operation is started immediately, otherwise the request is queued. Note that the I/O request is started by calling the *start* procedure variable that forms part of the *ControllerDesc* object. This procedure variable and the other procedure variables (*transfer, cleanup*) that form part of the *ControllerDesc* object, are called within the generic module. The specific module overrides

these procedures to provide a device specific implementation.

This method works well when device drivers must be written for a number of devices that form part of the same family and differ only in minor ways. However, the ideal case of a perfectly generic device driver that can be extended as necessary, is difficult to achieve. Development time for a generic driver is typically longer than for a similar non-generic driver, due to the additional work to identify and implement the generic and specific parts of the driver. However, subsequent specific modules that make use of the same generic module are simple to implement. This is the major advantage of extensible drivers. However, the extensible design makes the device driver more complex and difficult to understand and if the design is not generic enough, the generic module must be modified. This means that, once again, the developer must understand the whole device driver as well as its interaction with the underlying kernel. Extensible drivers can be implemented in user space [15] or kernel space [32].

2.2.3 Dynamically Loadable Device Drivers

A kernel that contains statically linked device drivers must be re-linked whenever it is configured for new peripheral devices. Clearly this makes it difficult to configure the kernel since the developer must have access to the complete kernel source code or the compiled object code. To alleviate this problem the kernel must support many different peripheral devices, but the unused drivers take up unnecessary space. A better solution is to allow device drivers to be specified at boot time [12]. The kernel then obtains the drivers from some storage space, usually from a local hard drive or over the network. Of course, the necessary drivers to access the hard drive or network must still be statically linked as part of the kernel. This makes it easier to configure the system, but once the system has booted, the device drivers are fixed in memory and cannot be replaced. It is therefore still necessary to reboot the system for each test run while the driver is being developed.

It is possible to make the process of loading device drivers more general by adding support for the run-time installation or replacement of code in the kernel. Mach 3.0 has support for installing trusted servers in the kernel [26] and the Linux kernel supports run-time loading and unloading of device drivers [3]. The SPIN system allows *untrusted* applications to download code into the kernel to change the interface and implementation of the operating system [5, 6]. However, such extensions must be written in a type safe language that also controls access to memory and privileged instructions. The Oberon system supports *dynamic loading* of modules and since a device driver in the Oberon system is just another module, it can be

loaded and unloaded at run time [41]. The Oberon system supports only one address space and depends on the Oberon language to protect sensitive parts of the system from incorrect applications [31].

Run-time loading of code can be used to develop device drivers that are dynamically loaded and replaced. This makes it possible to test a driver without having to reboot after each test run. A further advantage is that if a device is rarely used, the associated driver can be unloaded and its memory released. This saves memory, since only the devices that are in use, are loaded. When the device is used again, the operating system loads the driver automatically. This can be done without the knowledge or intervention of the users of the system.

2.2.4 User-Level Device Drivers

Micro-kernels like L3/L4, QNX and Mach 3.0 implement device drivers in user space [15, 20, 28, 30], thereby gaining address space protection and ease of configuration and development by sacrificing some efficiency. Since all communication between a client and a user-level driver is by way of interprocess communication (IPC), which is less efficient than a kernel call, special care has been take to optimize the IPC primitives of the above kernels.

The L3 micro-kernel was developed at the German National Research Center for Information Technology, with the aim of being a research kernel as well as a commercial system. The L3 kernel is written entirely in assembly language and implements only the most basic functionality that is needed to support the rest of the operating system. All the device drivers are implemented as user-level processes. This approach is claimed to improve the flexibility, extensibility and stability of the system, since these services are user processes that are protected from each other and can be debugged, unloaded and restarted like any other program [29, 30].

Each device driver process has its own address space that is extensible to include the I/O ports of the processor. I/O ports can be selectively assigned to a specific driver and a driver can be associated with a source of hardware interrupts. The L3 kernel converts hardware interrupts into messages that are sent to the associated driver. Figure 4 shows the structure of a generic device driver on the L3 kernel. It should be noted that the structure is similar to the device driver implementation in Minix as shown in Figure 1. This is not surprising, since the Minix device driver is also implemented as a process, even though it is implemented within the kernel itself.

User-level device drivers are prevented from corrupting the memory of other processes since

```
driver thread:
   do
      wait for (msg, sender);
      if sender = my hardware interrupt then
          read/write io ports;
          reset hardware interrupt
          ...
      else
          ...
      fi
   od
```

Figure 4: A generic L3 device driver

they execute within a protected address space. However, on the ix86 architecture, external DMA devices can override the memory protection mechanism of the processor and write directly to physical memory [11, 21]. This has potential security risks. Therefore DMA is restricted to trusted device drivers in L3.

The developers of L3 have found that their only serious performance problem was with a user-level RS232 driver, because this driver is activated each time a character is received [30]. The extra overhead of fielding an interrupt for each character at the user-level was significant. The problem was solved by implementing a small server within the kernel to buffer the incoming characters until a string of predefined length had been received. This whole string was then passed to the user-level driver for further processing. The overhead of switching to the user-level for each interrupt was therefore reduced, but at the cost of implementing a device dependent server within the kernel.

User-level device drivers have a number of distinct advantages:

- The developer works within a protected address space. This prevents a malfunctioning device driver from corrupting the memory of another process or the kernel. However, if privileged instructions are allowed, incorrect use of these instructions could still cause errors that will require a reboot. An alternative is to perform the necessary privileged instructions, for example access to the I/O ports, via kernel calls. The kernel can then ensure that the privileged instructions are used in a correct and safe manner, although this is less efficient.

- The developer interacts with the kernel via system calls and is therefore shielded from the complexities of the kernel.

- It is not necessary to reboot the kernel each time a modification is made to a user-level

device driver since it can be reloaded for each test run.

- The kernel can be configured for various environments without re-compilation.
- The same debugging tools that are used for user programs can be used without modification to debug the device driver.

There are also a number of disadvantages:

- The extra overhead of the context switch and message passing operations from the client thread to the user-level device driver makes the driver less efficient than in-kernel device drivers. However, it was shown that this loss of efficiency can be minimized to such an extent that it is acceptable in most cases [27].
- In-kernel device drivers can share common code and libraries, for example code to manage Direct Memory Access (DMA) transfers or code for timing. If each device driver is implemented in its own address space, these libraries must be duplicated. Furthermore, each instance of an address space has additional memory overhead, for example the page tables that define the address space.
- On the IBM PC hardware, mechanisms like DMA bypass the memory protection hardware. User-level drivers that use DMA must therefore be trusted.
- A user-level driver that can perform privileged instructions and has control over interrupts and I/O ports could bring the whole system to a halt.

User-level drivers can be used as an aid to the rapid development of in-kernel drivers. The driver is then developed in the protected user-space environment and after it has been stabilized, it is moved to the kernel to improve its efficiency. This method is only realistic if the implementation of device drivers is similar in both the kernel and user-space, and if the user-space driver does not use mechanisms that are not provided within the kernel.

2.3 The Gneiss Micro-kernel

The Gneiss micro-kernel was used as the test bed for the implementation of the different device driver development strategies that are examined as part of this thesis. To understand the design and implementation of these strategies, it is necessary first to examine the relevant details of the Gneiss kernel. The Gneiss kernel is a message based micro-kernel that supports

the client-server model and is aimed at the development of embedded software. It is an improved version of the HYBRID kernel and is implemented on the i386 hardware [16, 17].

2.3.1 Terminology

In operating system literature, there are often different definitions for the same concept. In most cases this is caused by a slight difference in implementation from one operating system to another and a need to emphasize this difference. In Unix, a user-level program that executes within its own address space and is scheduled by the kernel is called a *process*. Processes are expensive to create, because a new address space must be allocated and initialized for each new process. Another problem is that IPC between Unix-like processes is slow, because a full address space switch must be performed. These efficiency concerns were addressed with the introduction of a *light-weight process*. A light-weight process is a thread of execution that is scheduled by the kernel and can share an address space with other light-weight processes. The advantage of a light-weight process is that it is easier to create than a process and allows more efficient IPC via shared memory, although this complicates programming and debugging. Some kernels, like Mach, also support light-weight processes that are scheduled entirely in user-space. These are called threads to distinguish them from light-weight processes that are scheduled by the kernel [10]. However, there is some confusion in the literature about these concepts. For example, light-weight processes and threads are also described respectively as kernel supported threads and user-level threads, while processes are described as virtual machines, tasks or team-spaces. To avoid misunderstanding, the basic concepts relevant to the Gneiss micro-kernel will now be defined more precisely.

2.3.2 Virtual Machines and Threads in the Gneiss Kernel

In the Gneiss kernel, the environment for program execution is called a *Virtual Machine* (VM). A VM has its own address space and is an abstraction of the underlying physical machine. Each VM may support a number of light-weight processes that share the VM's address space and are scheduled by the kernel. These light-weight processes are called *threads*. The term thread was chosen because it is used in the *thread model vs. event model* comparison, which is a generally accepted expression, and this comparison is an important part of this thesis.

An executing virtual machine is active until one of two things happens. Firstly, a VM can execute until the time-slice allocated to it expires, when it is preempted by the kernel. The next VM can then be scheduled. Secondly, a VM can be descheduled when none of the threads

it contains is ready to execute. In such a case, the VM has not yet completed its time-slice, so the next VM to be scheduled can only use the part of the time-slice that was left over by the first VM, because a time-slice expires at a fixed interval. A better way to do this would be to reprogram the timer each time a new VM is scheduled. This would ensure that a VM would always be able to execute for a full time slice, but this would make the kernel more complex and incur additional overhead.

Unlike VMs, threads are scheduled non-preemptively, which means that a specific thread continues to execute until it voluntarily relinquishes control by performing a kernel call. However, there is no danger that a non-preemptive thread could cause the system to loop forever, since the time slice of the VM that contains it will eventually expire.

Non-preemptive scheduling is used for threads instead of preemptive scheduling because this simplifies the scheduler considerably and it removes the need to use explicit concurrency mechanisms. Since threads that share an address space can access the same memory locations, it is necessary to ensure that concurrent access to shared memory locations does not leave global data in an inconsistent state. If threads were scheduled preemptively, a thread could be preempted at any point and the programmer would have no way of predicting when a thread would lose control. This would mean that all critical data structures must be protected with synchronization primitives such as semaphores, critical sections or message passing. A programmer using non-preemptive threads has control over the descheduling points of the thread. This makes it easy to protect shared global variables since the programmer can ensure that control is released only when the shared variables are in a consistent state. A drawback is that the programmer must know the location of each possible descheduling point, since the program can exhibit inconsistent behaviour if a thread is descheduled unexpectedly, for example if a thread reaches a descheduling point as part of a library call.

Each thread within a VM appears to execute as a single deterministic flow of control. However, the nondeterministic scheduler ensures that after a thread has been descheduled, a programmer cannot predict which thread will be scheduled next. In the thread model each thread executes, at least in theory, concurrently with the other threads in a VM. Of course, on a single processor system this is just an illusion that is maintained because the scheduler ensures that the threads alternate so quickly that it seems they are being executed concurrently.

A VM has the following attributes:

- It has a protected virtual address space.
- A VM is preempted when its time-slice expires.
- A VM supports one or more threads of execution. A thread executes until it voluntarily relinquishes control by executing a kernel call. The kernel then schedules another thread in the VM for execution.
- It supports the non-privileged instruction set of the underlying physical machine.
- It has additional primitives to implement interprocess communication. These primitives are blocking and are used for sending, receiving and replying to a message.
- Only one thread can be active in a VM at a time.
- A thread can be dynamically created or destroyed.
- A VM is descheduled when none of the threads it supports is ready for execution.

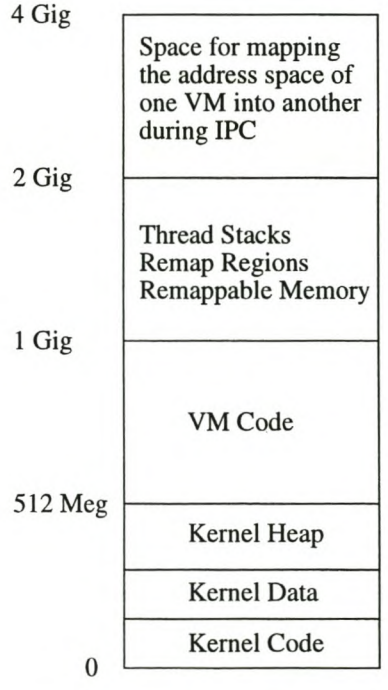

Figure 5: The memory layout of a Virtual Machine (VM)

2.3.3 Memory Model

To understand the memory model of the Gneiss kernel it is necessary to have a brief look at the memory management hardware of the Intel i386 architecture. The i386 has a two level memory management scheme of which the first part, called segmentation, consists of translating virtual addresses into linear addresses and an optional second part, called paging, consists of changing linear addresses into physical addresses.

In the segmentation model, virtual memory is organized as a number of units of variable size, called segments. These segments each have a base address which specifies the starting address of the segment in the linear address space and a limit, which is the largest offset that can be used within that segment. The hardware ensures that while a specific segment is active, any memory reference outside that segment will cause an exception.

Each segment can address up to 4 Gigabytes of memory and there must be at least one segment each for code, data and stack. In the Gneiss kernel each of these segments are initialised in such a way that they start at offset 0 of the linear address space and have a length of 4 Gigabytes. In this way, a flat memory model is constructed and since the segment registers are never changed after this, segmentation can for all intents and purposes be ignored from here on.

Segmentation must always be used, but an optional mechanism called paging can also be activated. Paging operates on fixed-sized pieces of memory that are called pages and each has a size of 4K. Figure 6 shows the layout of the paging system of the i386 processors. Like most modern processors, a 2-level paging system is used. The page directory has a size of 4K and contains 1024 entries of 4 bytes each. Each page directory entry contains the address of a page table, which like the page directory is 4K big and contains 1024 entries of 4 bytes each. These entries, along with a part of the linear address, are used to construct the address in physical memory of the page.

The i386 has four protection levels that range from the most privileged level, namely the supervisor level to the least privileged level, the user level. Certain instructions, like modifying the address of page directory, can only be executed at the supervisor level. A more privileged level can also be protected from unauthorized access by a less privileged level.

Figure 5 shows that the kernel is part of the address space of each VM, but since the pages that make up the kernel code execute at supervisor protection level they are not directly accessible to user code. Since the kernel is part of a VM, a context switch to the kernel due to a kernel call or hardware interrupt takes place within the same address space, even

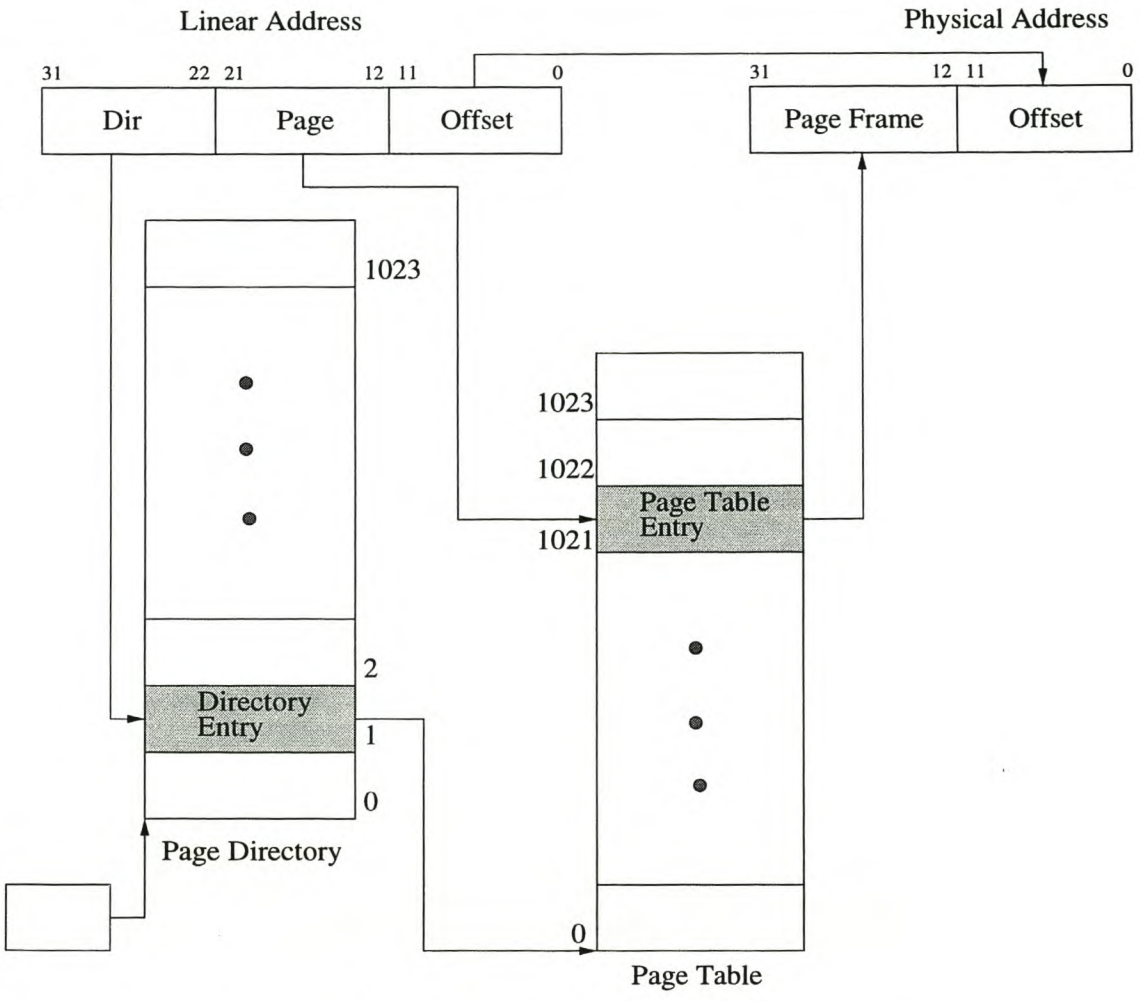

Figure 6: The paging system of the x86 processor

if the protection level is changed. This is more efficient than switching to a completely new task and it means that the kernel has full access to the address space of the VM. It is also possible for the kernel to access the address space of any VM on the machine by activating the appropriate page directory. Data can therefore be copied between any VM and the kernel in an efficient way.

Each thread in a VM has a user-level stack, but no equivalent in-kernel stack. Instead, in the kernel a single stack is shared by all the threads. This arrangement is made possible because interrupts are disabled while the kernel is executing. The kernel therefore forms a single critical section that does not need to be re-entrant[1]. This makes the kernel much simpler

[1]Hardware exceptions cause re-entry, but are handled as a special case

since it is not necessary to protect data structures from concurrent access. However, it is critical that the kernel executes for as short a time as possible to prevent the loss of data due to a failure to respond fast enough to an interrupt. The only time that the kernel executes with interrupts enabled, is when there are no threads that are ready for execution. The kernel then waits until an interrupt occurs.

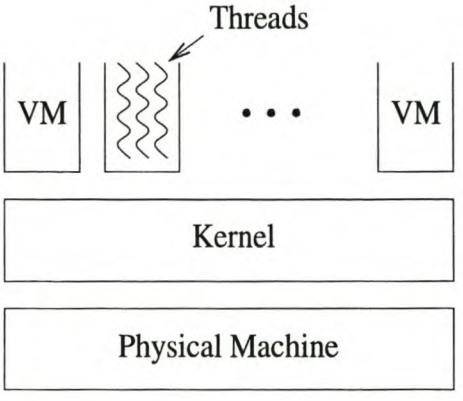

Figure 7: Multiple VMs on a single machine

A virtual machine is not an exact copy of the underlying hardware, but rather an abstraction to hide the intricate details of the physical hardware. Each machine can support multiple VMs that are time-sliced preemptively and are scheduled in a round-robin fashion.

2.3.4 Interprocess Communication

Since VMs do not share memory, threads in different VMs must exchange messages to communicate. The kernel supplies three interprocess communication (IPC) primitives: *Transaction*, *ReceiveRequest* and *SendReply*. These primitives have the following attributes:

- Communication is done via an abstract channel called an IPC port. A port decouples the sender and receiver threads from each other, since it removes the need for the sender to specify the receiver directly. Ports must be registered with the kernel and belong to a name space that is visible to all the VMs on a physical computer. This allows a thread in any VM on the computer to communicate with a thread in any other VM.
- The IPC primitives are blocking and provide unbuffered message passing.

- An IPC message consists of a fixed length *header* that contains parameters and a variable length *buffer* which contains the data pertaining to the operation.

A client thread uses the *Transaction* primitive to send a message. The *ReceiveRequest* primitive is used by a server thread to receive a message and the server must then use the *SendReply* primitive to reply to the message from the client. The IPC mechanism is synchronous since the client thread blocks until the server has replied. For the sake of simplicity, nested *ReceiveRequests* are not allowed.

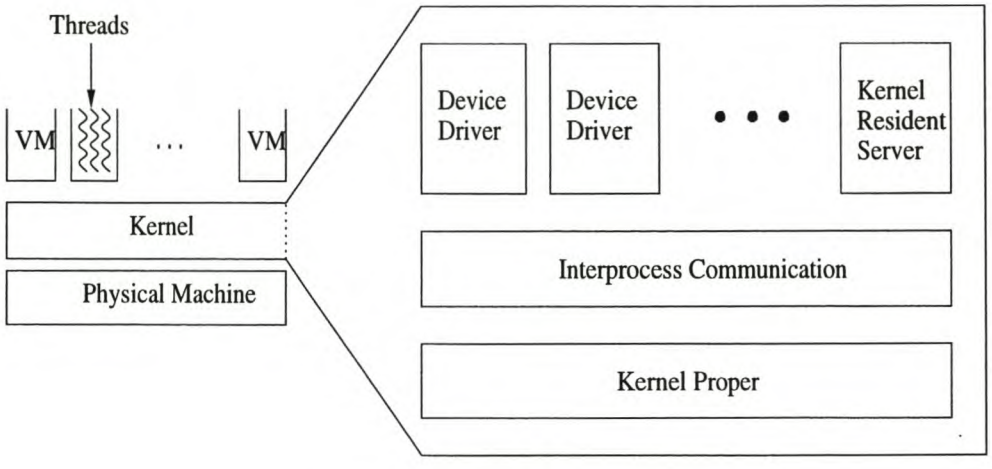

Figure 8: The Kernel Structure

2.3.5 Kernel Resident Servers

According to the Gneiss design philosophy, VMs support either clients or servers. A server can also act as a client by performing a client request to another server. Servers can be implemented in user space or within the kernel as kernel resident servers. As the name suggests, kernel resident servers execute within kernel space and provide services, for example thread and VM creation, to user-level clients. Initially, device drivers in the Gneiss kernel were implemented as kernel resident servers, but in this thesis alternative techniques are explored. A kernel-resident driver executes at the same privilege level as the rest of the kernel and has access to the address spaces of the kernel and all other VMs. These drivers can also make use of libraries within the kernel to perform common operations, for example interrupt handling and I/O operations.

When a thread in a VM needs to access a kernel resident server it makes use of the *Transaction*

primitive. Since the *Transaction* primitive is also used to pass messages to other threads, the kernel examines the parameters of the *Transaction* call to determine whether it is aimed at a kernel resident server or at another thread. If the intended target is a kernel resident server, a procedure call is performed to a handler that was installed by that server. This means that an operation that is implemented in a kernel resident server can be executed faster than if the same operation was implemented in another VM, since the extra context switch and message passing overhead is eliminated. A client thread does not need to know whether a server is implemented within the kernel or in another VM.

2.3.6 A Typical Kernel Resident Device Driver

The structure of a typical kernel resident device driver is shown in Figure 9. This structure influenced the design of Event Driven VMs, which were developed to support the implementation of device drivers in user-space. The design and implementation of Event Driven VMs are examined in the next chapter.

The device driver in Figure 9 receives a stream of events. These events are either client requests, interrupts, or timeouts that have expired. The events are handled by event handlers that are installed by the device driver either when it is initialized or, in the case of timeouts, before a long-running device operation is started. The initialization section registers the handler and binds the interrupt handler to a specific interrupt and the transaction handler to an IPC port. Client requests are processed within the *TransactionHandler* procedure. In case of a time consuming device operation, the client thread is blocked after the operation is started and will only be resumed after the device operation has been completed. It is not necessary to block the client thread if it is possible to complete the request without having to wait for the peripheral device, for example if the requested information already is available in a cache. An *InterruptHandler* is bound to a specific hardware interrupt and is called when that interrupt is triggered. If the interrupt signalled the completion of the current device operation, the client thread that was blocked on that operation can be resumed. A timeout is set to guard against the possibility that a defective device could fail to send an I/O completion interrupt. If the device operation completes successfully, the timeout is cancelled within the interrupt handler. However, if the selected interval passes before the interrupt handler is called, the *TimeoutHandler* procedure is called. Appropriate recovery can then be done within the timeout handler.

Since kernel-resident servers are implemented within the kernel, the same advantages and disadvantages that are outlined in Section 2.2.1 apply to them.

```
MODULE DeviceDriver;

(* Called when an interrupt is triggered *)
PROCEDURE InterruptHandler;
BEGIN
   Handle interrupt and copy data to client buffer;
   Cancel timeout;
   Resume client thread
END InterruptHandler;

(* Called when a device request takes too long to complete *)
PROCEDURE TimeoutHandler;
   Abort device request;
   Resume the client thread with the relevant error information;
END TimeoutHandler;

(* Handle client requests, that are issued with the Transaction primitive *)
PROCEDURE TransactionHandler;
BEGIN
   CASE operation OF
      read: IF request can be satisfied immediately THEN
               perform operation and return
            ELSE
               block executing thread and start device request;
               set timeout(TimeoutHandler)
            END
     |write: ...
   END
END

BEGIN
   Install InterruptHandler;
   Install TransactionHandler
END DeviceDriver.
```

Figure 9: A typical kernel-resident device driver

2.4 Summary

This chapter provided an overview of the various properties of device drivers. The effects of the structure of the operating system kernel and kernel support on device drivers have also been discussed. A number of well known micro-kernels that implement device drivers in user space have been examined and the Gneiss micro-kernel that was used as a test-bed for the implementation has been examined in some detail. The next chapter will describe how the Gneiss kernel was modified to support user-level device drivers.

Chapter 3

User-Level Device Drivers

The previous version of the Gneiss kernel had support for the development of in-kernel drivers and extensible drivers, but did not provide support for either user-level drivers or loadable drivers. It was decided to extend the kernel to make the development of user-level drivers possible. The first step was to forward interrupts to threads. This was done by implementing a small in-kernel server to field the interrupts and forward them to the relevant thread. Support was also added to modify the I/O privilege of a specific thread to allow the I/O ports to be accessed.

In the original design of the Gneiss kernel, no nested *ReceiveRequest* calls were allowed. This was done for the sake of simplicity and to improve performance. However, this makes it difficult to implement a user-level device driver that schedules incoming requests according to some scheduling criteria, because a thread must reply to a message before the next message can be received. Therefore all requests must be handled on a first-in-first-out basis. An alternative solution is to use a *selected receive* primitive that allow messages to be received only if they satisfy certain conditions. However, *selected receive* was not used because it is too restrictive and is difficult to implement efficiently. The problem was solved by using synchronization primitives [1] to allow individual threads to be blocked and resumed.

The thread model is not necessarily the most suitable model for the development of device drivers. The event model was considered as an alternative, because of the event driven nature of device drivers and also because the event model and the thread model are often compared. A number of changes were needed to extend the kernel so that it could support the event model. The rest of this chapter examines the design and implementation of these changes.

[1] These primitives have been implemented by Johan de Villiers

3.1 The Event Based Programming Model

An event is defined as an operation or signal, from outside the system that is being studied, that occurs asynchronously and that requires urgent attention. Hardware interrupts, software messages and page faults are all examples of events that can influence a user program. The act of triggering the event is called *raising* the event. Events that are raised by the operating system or the hardware, for example page faults and interrupts, are called *system events*, while events that are explicitly raised by a user program are called *user events*.

When an event is raised, it eventually results in the execution of some piece of code. This piece of code is called an *event handler*. There can be more than one recipient that is interested in a particular event, which means that more than one event handler could be executed when an event is raised. The process of selecting the appropriate event handler to which an event should be dispatched, is called *event dispatching*.

The event model is primarily non-deterministic, since the thread of control can be transferred to an event handler without the programmer being able to predict when it will happen or in what order the events will be raised. An event handler usually executes quickly and returns control to the system after a short time. This gives the impression of multiprogramming, although it is not true concurrency.

Events can be classified as *blocking* or *non-blocking*, depending on how the event is raised. An event is called *blocking* if it must be handled to completion before another event of that type can be raised, for example an event that is raised due to a page fault. A *non-blocking* event is raised irrespectively of whether an event of that type is already being handled, for example an event that is raised due to timer notification.

Events arrive asynchronously, which means that no predictions can be made about when and in what order they will arrive. Programs that are implemented to support the event model must be able to respond to these external events. The flow of control of an event driven program is therefore primarily determined by the arrival of events. For the sake of simplicity, an event is usually handled to completion before the next event is accepted. This means that events must be queued until the handler is ready to respond to them. However, sometimes it is preferable that a high priority event be handled immediately. This will cause the executing event handler to be interrupted until the higher priority event has been processed.

Device drivers fit well into the event paradigm, since they are servers that must respond to events such as interrupts that can occur without warning and must be processed as quickly

as possible.

3.1.1 Design Aims for User-Level Device Drivers

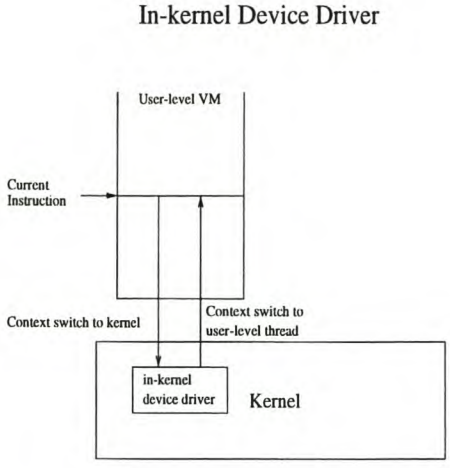

Figure 10: This figure shows a thread that is interrupted. A context switch is then performed to the kernel, where the interrupt is handled by an in-kernel device driver. After completion of the interrupt handling the thread is resumed.

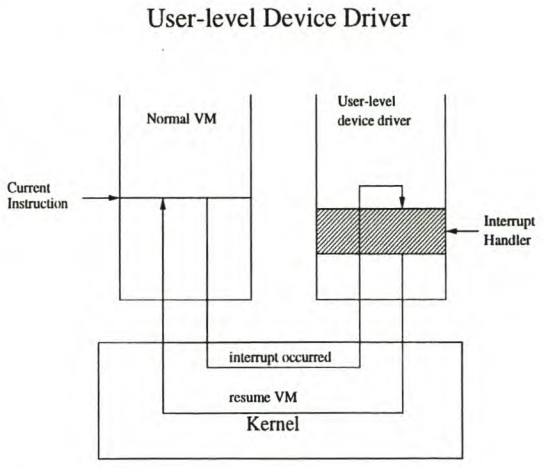

Figure 11: This figure shows how a user-level device driver is inherently less efficient than an in-kernel device driver, due to the extra context switch to user space and back to the kernel.

To develop a model that supports the development of user-level device drivers, it is first necessary to examine the desirable properties for such a driver. These properties are:

- **Efficiency:** Special care must be taken to ensure that the performance degradation due to cross address space context switching and message passing is minimized. Figure 10 shows a thread that is executing when an interrupt occurs. A context switch is performed to the kernel where the interrupt is handled by an in-kernel device driver. Thereafter a context switch to user-space is performed to resume the execution of the thread. In Figure 11 the device driver is implemented in user space. If an interrupt occurs while a thread is executing, two context switches must be performed. The first context switch is to the kernel and the second context switch is from the kernel to the user-level device driver. After completion of the user-level interrupt handler, control returns to the interrupted thread via the kernel. These extra context switches and the associated Translation Lookaside Buffer (TLB) flushes make user-level device drivers inherently less efficient than in-kernel drivers.

- **Access to privileged operations:** A user-level driver must have access to I/O ports and interrupts and must be able to initiate Direct Memory Access (DMA) operations.
- **Safety and security:** A user-level driver executes within its own address space and is therefore isolated from the rest of the system. However, malicious or incorrect use of the privileged operations that are mentioned above could still compromise system security or cause system starvation. System starvation can occur when the user-level driver switches interrupts off and executes an endless loop. Since the user-level driver cannot be interrupted, neither the kernel nor the other user-level threads can get a chance to execute. This danger must be minimized if the user-level drivers are trusted or completely eliminated if the user-level drivers are untrusted.
- **Ease of use:** A user-level driver must be much easier to develop, debug and maintain than an in-kernel driver to offset the inherent performance penalty compared to an in-kernel driver. It must be easy and efficient to develop shareable non-polling device drivers without any elaborate work-arounds.
- **Ease of relocation:** Hardware that generates an excessive number of interrupts could cause a user-level driver to operate at an unacceptably low level of efficiency, since a context switch to the user-level driver must be performed for each interrupt. This problem can be addressed by relocating the driver to the kernel. Therefore it must be easy to re-implement a user-level driver as an in-kernel driver. This approach can be used as a general development practice, which entails that the driver is developed in user-space for added protection and ease of development and thereafter moved to the kernel for reasons of efficiency.
- **Transparency:** The interface to the device driver must be implemented in such a way that it is unnecessary for a client to know whether a request is addressed to a user-level driver or an in-kernel driver. It must be possible to move a driver between the kernel and user-space without modification to the client.

3.2 Event Driven VMs on the Gneiss Kernel

Instead of extending the existing Virtual Machine (VM) abstraction to support both the thread model and the event model, a new abstraction called an Event Driven Virtual Machine (EDVM) has been designed to support the event model. This decision has the advantage that the event and thread paradigms are not mixed which made it easier to compare them and it also makes the addition of the event model less intrusive to the existing system.

An EDVM is in many respects similar to a VM: it has a protected address space, it can be preempted if it executes for too long and it has primitives for interprocess communication. However, there are some crucial differences: the basic unit of execution is an event handler that is only executed when a relevant event is raised, unlike a thread in a VM that is scheduled by the kernel. An EDVM could contain event handlers for events that are raised due to hardware interrupts, client IPC operations or timeouts that have been triggered. An event executes until completion, except when it is interrupted by an event that must be dispatched to a handler in a different EDVM.

In the next sections the design and operation of an EDVM and the various problems that had to be overcome will be discussed in more detail.

3.2.1 The Design of an EDVM

As described in Section 2.3, the Gneiss kernel has been designed to support the thread model and client-server applications. As a result of these design decisions, a number of constraints were enforced when the kernel was modified to support EDVMs. These constraints are:

- **Blocking Kernel Calls** make an event handler less responsive. If a kernel call is performed by an event handler, the handler is blocked until the kernel call has completed. During that time, the handler is not available to process any events that are directed at it.
- **Synchronous Message Passing** also makes an event handler less responsive, since message passing between different protection domains is done by using blocking kernel calls. Since all message passing on the Gneiss kernel is synchronous, it is desirable that messages to an EDVM also be synchronous for the sake of consistency.
- The kernel does not provide support for **critical sections** which means that an interrupt event handler could be preempted when a EDVM's time-slice expires, to be resumed only when the EDVM is scheduled again. This could cause problems in interrupt handlers which must meet deadlines that are enforced by peripheral device hardware.

With the above constraints in mind, an EDVM has been designed to have the following properties:

- It has a protected address space. This prevents a malfunctioning device driver in an EDVM from corrupting the memory of the rest of the system.

- It supports the installation of multiple event handlers that are executed only when the relevant event is raised.
- It supports the basic non-privileged instruction set of the underlying physical machine, as well as providing control over interrupts and I/O operations.
- It supports the *Transaction* primitive, which can be used to request services from thread based VMs. In contrast with thread based VMs, the *ReceiveRequest* and *SendReply* primitives are not supported, since the kernel converts any messages that are passed to an EDVM into events.
- It does not contain any threads that are scheduled by the kernel. An event handler is executed after an event is raised and dispatched to that handler. The event handlers in an EDVM are therefore scheduled by the events that they handle.
- It does not support dynamic thread creation, since it contains only event handlers.
- An event handler executes until completion to enforce mutual exclusion of handlers. However, if an event handler is interrupted by a higher priority event that is targeted at another EDVM, the executing handler is preempted and the higher priority event handler is activated.
- If an event handler executes for a long time, the time-slice of the EDVM will expire. The EDVM is then suspended and all events that are directed at it are entered into a queue. This is necessary to maintain mutual exclusion between the event handler that was executing when the EDVM was time-sliced and the other event handlers in the EDVM.
- Events can be blocked, to be resumed at a later stage.

3.2.2 Communication between Thread VMs and EDVMs

If the event model and the thread model are to co-exist successfully on the Gneiss kernel, it is essential that clients that use the thread model can send requests to an EDVM server and vice versa. In some systems, like QNX, the raiser of the event is not interested in interacting with the receiver and does not wait for a reply to the event [20]. According to this model, an event is an announcement that does not require a response. However, a thread in the Gneiss kernel uses synchronous message passing, which means that the thread is blocked until it has received a response. One strategy that was examined was to extend thread VMs to allow

asynchronous message passing operations to be used. Asynchronous message passing has a number of advantages, for example it is easier to implement certain algorithms like heartbeat and broadcast algorithms [2]. However, a drawback of asynchronous message passing is that it is more difficult to recover when an error has occurred. For example, the sending thread could send data much faster than the receiving thread can process it and the only way of verifying that the receiver received the message is for the sender to wait for a reply message. This is inefficient, because extra messages must be sent. Another design aim was to keep the syntax of message passing primitives the same, irrespective of whether the client is communicating with a thread in a VM or an event handler in an EDVM. This means that the client does not need to know whether the server is in fact an EDVM or a thread VM. For these reasons it was decided not to implement asynchronous message passing, but instead to use synchronous message passing.

This design decision means that EDVMs must support blocking events, since the client thread that raised the event must be blocked until the request is completed. When a client thread sends an IPC message to an EDVM, the kernel transforms the message to an event that is then dispatched to the relevant event handler. The semantics of the *Transaction* message passing primitive therefore remains unchanged, although the implementation is similar to a *remote procedure call* rather than a *rendezvous*. The difference between a *remote procedure call* and a *rendezvous* is that for a *remote procedure call*, a new thread is created, at least in concept, to service the request, while a *rendezvous* is a synchronization between two existing, independently executing threads. Although the event handler is not created especially to service the event, it does not execute on its own, but waits until an event is raised. This makes the implementation similar to a *remote procedure call*. An important benefit of blocking events is that information can be returned to the client thread after the event handler has finished execution.

3.2.3 Categories of Event Handlers

A number of categories of event handlers have been defined because the format of the data that is associated with an event differs according to how the event was raised. For example, if an event is raised due to a hardware interrupt, the Interrupt Request number (IRQ) is needed, while if an event is raised due to message passing, the operation must be accompanied by the message data. It is desirable that the various events and their associated data will be handled in a consistent manner, since this will simplify development under the event model.

To achieve this aim a number of mechanisms were examined. For the first mechanism a

procedure interface is defined for the handler of each type of event. A stack frame is then built up in the kernel so that when control is transferred to the handler, it is as if the procedure is called with the relevant parameters. The disadvantage of this approach is that it is a compiler specific operation to set up the stack frame manually and therefore it is not portable. This mechanism was refined by defining a record structure for each type of event. The fields of each of these records are divided into two parts 1) the common fields that are generic to all events and 2) the specific fields that are unique to an event. It is still necessary to set up a stack frame, but because only one parameter, namely a pointer to the record, is now needed it is much easier to set up the stack frame. Another advantage of this approach is the drivers that are developed in this way are similar to in-kernel drivers. This makes it easy to relocate drivers between the kernel and user-space.

The different types of events are classified into the following categories:

- **Interrupt Events** are raised by hardware interrupt sources. When a hardware interrupt occurs, control is immediately transferred to the kernel. The kernel then determines whether an event handler has been installed for that particular interrupt and if that is the case, the interrupt is converted to an event and dispatched to the handler. After the the interrupt event handler has completed, the interrupted thread is resumed. This ensures that the non-preemptive properties of threads are retained. An interrupt event can be blocking or non-blocking, depending on the hardware interrupt source. Timer interrupts are non-blocking, while interrupts that must be acknowledged before another interrupt can be raised, are blocking. An interrupt event is a system event.

- **Transaction Events** are raised when a client thread uses the *Transaction* primitive to send a message to an EDVM. As with *Interrupt Events*, the kernel first determines whether a *Transaction* event handler has been installed before converting the message into an event and dispatching it to the handler. After the event has been handled to completion, control returns the the client VM, but unlike when an interrupt event is raised, the client thread is not always resumed immediately. The scheduler is first invoked to determine whether there are other threads in the client VM that are ready to execute. This does not create problems since the execution of a message passing primitive serves as a descheduling point. A transaction event is a blocking user event.

- **Signal Events** are used for non-blocking signalling between different event handlers. A signal event is therefore a non-blocking user event.

- **Timeout Events** are raised by the timer hardware, but the timeout is set by the

EDVM. If the timeout is not cancelled within the allotted time, it expires and the *Timeout* event handler is called. A timeout event is a non-blocking system event.

Figure 12 shows a simple example of how a *Transaction* event handler is installed and used.

```
MODULE GenericDeviceDriver;

CONST
   PortNum = 511;
   Read = 0;
   Write = 1;

TYPE
   TransactionEvent = POINTER TO TransactionEventDesc;
   TransactionEventDesc = RECORD
      header: POINTER TO ARRAY 16 OF LONGINT; (* for parameters *)
      buf: LONGINT;            (* a pointer to the data buffer *)
      reqlen, replen: LONGINT; (* the request and reply length of the buffer *)
   END;

(* A handler that is called by the kernel to handle a Transaction event *)
PROCEDURE TransactionHandlerProc(event: TransactionEvent);
VAR
   operation: LONGINT;
BEGIN
   (* extract parameters from the header *)
   operation := event.header[2];
   CASE operation OF
      Read:
         (* Perform device operation to read data into the
           buffer pointed to by event.buf *)
         ReadData(event.buf, event.replen);
     |Write: (* Write data to device *)
     |GetDeviceParameters: (* get device information *)
   END
END TransactionHandlerProc;

BEGIN
   InstallTransactionHandler(TransactionHandlerProc, PortNum)
END GenericDeviceDriver.
```

Figure 12: A sample program that shows how a TransactionHandler is installed

3.2.4 Mutual Exclusion of Event Handlers

If an interrupt event is raised while a thread in a VM is executing, the response is simple: control is transferred to the kernel where the executing thread is blocked and the event is dispatched to the relevant handler. However, if an event handler was executing when another event was raised, there are a number of ways in which the second event can be handled. One

possibility is to interrupt the executing event handler and transfer control to the new handler. After the new handler has finished execution, the interrupted handler is then resumed. If this method is used, a mutual exclusion mechanism is needed to allow event handlers to update critical data structures without the danger of being interrupted. There are a number of mutual exclusion mechanisms that can be used for this purpose, for example, critical sections and semaphores. However, these mechanisms place an additional burden on the programmer who must then determine which data structures should be protected and who must also ensure that the mutual exclusion primitives are used correctly. It is not difficult to implement these protection mechanisms, but since they can be difficult to use correctly they have been deemed unsuitable.

It is possible to remove the need for explicit mutual exclusion mechanisms completely by restricting event handlers to handle only one event at a time. This means that the handler executes to completion before the next event can be handled. If an event is dispatched to a handler that is busy executing, the event must be queued until the handler has completed. A disadvantage of this approach is that event handlers that run for a long time will reduce the responsiveness of an EDVM. An event handler should therefore complete quickly. An optimisation is possible if an event is raised with a higher priority and that event is dispatched to a different EDVM than that of the handler that is currently executing. In that case the executing handler is interrupted and control is transferred to the higher priority handler. After the higher priority handler has completed, the interrupted handler is resumed immediately. This optimisation causes no mutual exclusion problems, because the handlers are in different protection domains.

3.2.5 Blocking a Transaction Event

When an event driven device driver receives a transaction event it contains in many cases a request from a client to perform some device operation. This operation could take a while to complete and during this time the event handler cannot handle other events. Clearly this is not acceptable, since it prevents the driver from handling more than one user request at a time. To combat this problem support was added to allow an event to be blocked, whereupon the event handler is again available to handle other events.

Before an event can be blocked, its associated data must be copied to global variables, since the stack of the event handler procedure is needed to handle the next event. This is inefficient if the event contains a lot of data, for example a large buffer that must be transferred to the peripheral device. The kernel therefore allocates a buffer for the associated data and maps

this buffer into the EDVM's address space. A pointer to the data is passed along with the event, instead of copying all the data to the stack of the event handler. The EDVM then has access to the message data of each blocked transaction event. This makes it possible to perform optimisations on the queue of blocked requests, like reordering the sequence in which device operations are performed. After a device operation has been completed, the blocked event is resumed and the client thread that issued the request can be scheduled for execution. This aids the development of shareable device drivers.

3.2.6 Reentrancy concerns

An event handler can be called reentrant if it can be interrupted to handle a second event while it is already processing the first event. This means that the state of the event handler at the point of interruption has to be stored so that the first event can be resumed after the second event has completed. The state that must be saved consists of the registers as well as the run-time stack.

If events are always handled to completion it is not necessary for the handler to be be reentrant. However, this approach causes problems when an event handler performs a blocking kernel call. Since the handler must execute to completion, it is unavailable to process other events until the kernel call has been completed. This is undesirable since it reduces the responsiveness of the event handler.

A possible approach is to disallow kernel calls from an event handler, but this would reduce the usefulness of EDVMs. An alternative approach is to add support for non-blocking kernel calls to the kernel. The responsiveness of an event handler could then be improved by executing only these non-blocking kernel calls. Unfortunately this means that the programmer must distinguish between blocking and non-blocking kernel calls, which makes EDVMs more difficult to use. Another disadvantage is that the kernel was not designed to support non-blocking kernel calls and the addition of such support would entail significant modifications. This approach has therefore been deemed unsuitable for the Gneiss kernel. It should be noted that both the above disadvantages are negated if a kernel was designed from the start to support non-blocking kernel calls.

The approach that was selected is to create an execution context for each category of event handler that is supported by an EDVM. The execution context consists of a register storage area and a run-time user-level stack and is used to execute the code of the event handler. When an event handler executes a kernel call and is subsequently blocked, events that are

directed at an event handler of another category can still be handled. For example, if a *Transaction* event handler is blocked, the *Interrupt* event handler is still available to process events. This does not affect the mutual exclusion of event handlers, since by performing a kernel call, the blocked handler showed a willingness to relinquish control.

3.2.7 Binding an Event to a Handler

After an event is raised, it must be dispatched to the relevant event handler. However, before this can happen, the handler must be identified in some way. This means that there must be a binding between an event and a handler.

It is simple to perform event dispatching in a single address space system, since a procedure call is all that is needed to dispatch an event to a handler. However, in the Gneiss kernel each VM executes in its own protection domain which makes event dispatching more complex. The kernel is part of each VM, because it is mapped into the bottom 64 Mbytes of each VM's address space. This means that an in-kernel device driver is always available when an interrupt occurs or when a client thread issues a request. Such a driver could also gain access to the address space of any VM by means of manipulating the page tables. This allows in-kernel device drivers to transfer data between the kernel and any VM with ease. However, a device driver in an EDVM executes in a different address space from that of the client thread. When an event is raised, control cannot pass directly to the relevant event handler, since this would prevent the internal data structures of the kernel from being updated. Instead, control is first transferred to the kernel and from there to the relevant handler.

In many event based systems like the Oberon System [41] and X-Windows [4, 24, 33], events can be propagated through a hierarchy of event handlers until the appropriate handler is eventually found to process the event. This method allows many events to be handled by default handlers and saves a user the effort of developing such handlers himself. The Oberon System also makes provision for an event to be bound to many event handlers that are each invoked in turn when the relevant event is raised.

In the Gneiss kernel, transaction events are blocking events as well as bi-directional events, since data can be returned to the client that raised the event. The bi-directional property of transaction events creates problems when more than one event handler is bound to an event. This is because although there is more than one target for the transaction event, there is only one source. Therefore if each target event handler returns data to the single source, it would make it difficult to choose the correct return data. A possible solution is to allow only one

event handler to return data. However, this puts an additional burden on the programmer, who might not even be aware of the other event handlers. Another solution would be to split the raising of the transaction event into two primitives, namely one that sends data and one that receives data. The receive primitive can then be called as many times as necessary to receive a reply from each target. However, this would add complexity and make it inefficient to send data if the sender must wait for a reply before more data can be sent, since two kernel calls will be needed instead of one.

The advantages of the solutions considered in the previous paragraph are outweighed by their respective disadvantages. For this reason, another approach was followed, namely that an event can be bound to only one handler at a time. An attempt to install a second handler for an event will result in an error. This direct mapping simplifies the identification and dispatching of an event to the relevant event handler. It also improves the efficiency of event dispatching, since a single event cannot cause the invocation of a number of event handlers that are possibly situated in different protection domains. The drawback of this method is that it is more difficult to broadcast an event to a number of recipients. However, this is not a serious problem since in the case of device drivers, an event is normally only relevant to a specific handler. For example, a hardware interrupt event is only relevant to the driver for the device that raised the interrupt.

3.2.8 Time-slicing an EDVM

An important cornerstone of the model of an event handler that executes until completion is that the handler should execute for as short a time as possible. However, there is nothing to prevent a handler from executing for an arbitrarily long time. Such a handler cannot be allowed to continue unchecked because it would prevent the rest of the system from getting a chance to execute.

One way of handling this problem is to set a timer when the event handler is invoked and cancel the timer upon completion of the handler. If the timer expired while the handler is executing, the handler is terminated. Unfortunately it is difficult to judge the correct interval for the timer, which means that a long running event handler could be terminated just before it would have completed execution anyway. A handler that is terminated while it is busy executing could also leave the whole EDVM in an inconsistent state since it could have been busy updating critical data structures. Due to these difficulties it is not a feasible solution to terminate the event handler after an interval.

A better approach is to preempt the EDVM if an event handler executes for a long time. Unlike VMs that support the thread model, an EDVM is normally not scheduled since the event handlers are triggered on demand. Therefore an EDVM is only preempted when an event handler executes until the EDVM's time-slice expires. The whole EDVM is then locked which means any events that are directed at it, are queued instead of being dispatched to event handlers. This is necessary to achieve mutual exclusion, since the time-sliced event has not completed execution and concurrent access could cause global data structures to become inconsistent. All events that are directed at a locked EDVM are queued and are dispatched to the relevant handlers when the EDVM is unlocked.

After being locked, the EDVM is added to the back of the queue that contains all the VMs that are ready to execute. This allows other VMs in the system to be scheduled. The EDVM will eventually be scheduled for execution and it is then unlocked. The event handler that was preempted then continues further from the point of preemption. In this way long-running event handlers are prevented from monopolizing the system.

The time-slicing of VMs and EDVMs are enforced by the system clock which generates interrupts at a fixed rate. This fixed interval time-slicing has certain limitations. When an event is raised, the remaining part of the client thread's time-slice is donated to the EDVM. This happens irrespective of whether a user event or system event was raised. Therefore an EDVM never has a full time-slice. It is conceivable that a handler could be invoked just before the EDVM's time-slice expires. In that case the event handler may not be able to complete execution before the EDVM is preempted by the timer, which could cause erratic response times. Therefore an EDVM is not preempted immediately when its time-slice expires. Instead, control returns to the event handler that was executing and the EDVM is only locked if the same handler is still executing when the EDVM is time-sliced again.

3.2.9 User Control Over Interrupts and I/O Ports

One of the properties of an EDVM is that it supports privileged operations like control over interrupts and I/O ports. A device driver could therefore mask interrupts either intentionally or by accident. This would prevent the timer interrupt from being triggered and the kernel would not regain control until interrupts are unmasked. The I/O ports can also be used to re-program important parts of the system, for example the keyboard controller or the system timer. Although uncontrolled access to the interrupts and I/O ports could have dangerous consequences, it is unnecessary and inefficient to have elaborate safeguards if a user-level driver could be trusted to use these privileged operations correctly. However, it would be

beneficial to make it impossible for an untrusted user-level driver to disrupt the system either by accident or by design.

One way in which this can be achieved is by allowing an EDVM neither control over interrupts nor access to I/O ports. Interrupt events can then still be dispatched to the user-level driver, but the driver will not be able to mask any interrupts or manipulate the I/O ports directly. Instead, the only way in which I/O ports can be accessed is via kernel calls. This means that the kernel can prevent an EDVM from accessing critical I/O ports, but at the price of the extra overhead that is incurred by the kernel calls.

Fortunately an alternative strategy is possible because the Intel i386 processors and its successors allow selected I/O ports to be added to the address space of a user-level EDVM while still preventing the EDVM from masking interrupts. This technique can also be used to control access to I/O ports and in this way prevent two different EDVMs from being able to access the same I/O port.

3.2.10 Event Dispatching

Events must be serviced quickly, especially hardware interrupt and exception events. A failure to do so can cause information to be lost. For example, if a network packet arrives while another packet is still being processed, the second packet must be dropped. Techniques like data buffering can help to a degree, especially if the data arrives in bursts, but if the processing of the data is simply too slow, the problem is only postponed.

State Transition Diagram

Figure 13 shows the state transition diagram for an event handler. A handler is always in one of the following states:

- **Running** - The event handler is executing.
- **Interrupted** - The event handler has been interrupted by a hardware interrupt. This can be an interrupt that is handled by another event handler or by an in-kernel device driver. The interrupt can also signify that the time-slice of the EDVM has expired.
- **Locked** - An event handler is locked when the time-slice of the EDVM has expired. This prevents concurrent access to shared data structures.

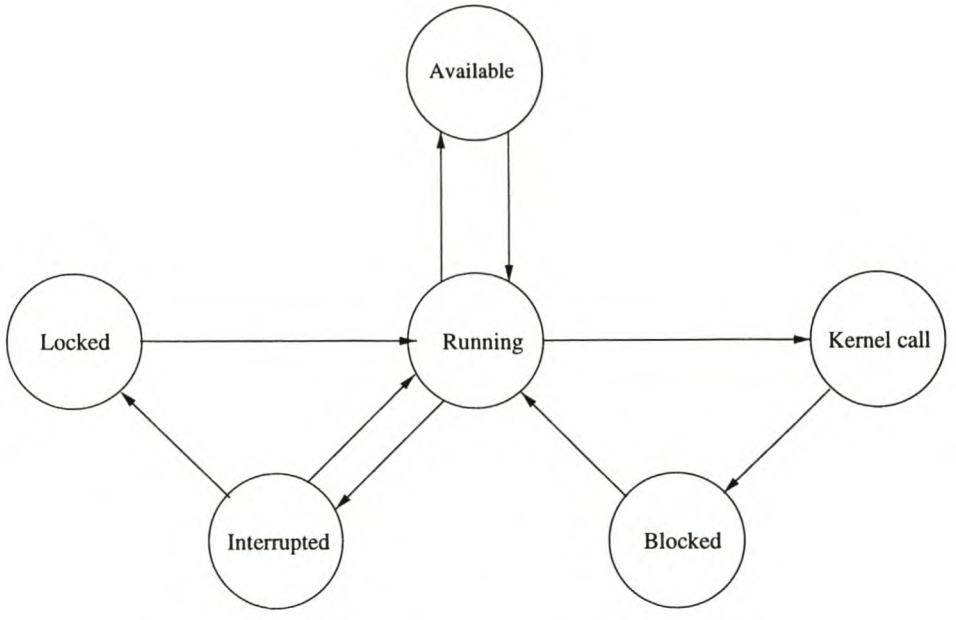

Figure 13: Event handler state transition diagram

- **Kernel call** - The handler is executing a kernel call.
- **Blocked** - The handler is blocked within the kernel.
- **Available** - The handler is available to execute an event that is dispatched to it.

Combining the Thread model and the Event model

It is more difficult to support both the event model and the thread model in a system than to support either of these models alone. This is because when both the thread model and the event model are supported, it is not only necessary to make provision for the working of each separate model, but also necessary to make provision for how they interact with each other.

In the case of the Gneiss kernel, support for Event Driven VMs was added to a micro-kernel that supported only the thread model and synchronous message passing. This created a number of problems because EDVMs execute under different assumptions and objectives than normal thread VMs. For example, the way in which VMs and EDVMs are scheduled is affected by whether a thread or event handler was executing when an event was raised or by whether the target of a message passing operation is a thread or an event handler. In the next sections the interaction between Event Driven VMs and thread VMs will be examined.

- **VM client to EDVM server**

  When an event is raised while a thread in a VM is executing, the thread is blocked and control is immediately passed via the kernel to the relevant event handler. This allows an EDVM to achieve a good response time while handling events, which is especially important when handling interrupt events.

  The following rules describe the scheduling after an event has been serviced:

  1. After a user event has been handled, the thread that raised the user event is descheduled if there is another thread in that VM that is ready to execute. This is acceptable because the thread showed a willingness to relinquish control when it raised the user event.
  2. In the case of a system event, the interrupted thread resumes executing without any rescheduling to maintain the model of mutual exclusion of threads. Timeout events have a minimum resolution of 20 milliseconds, which is the same as the length of a time slice. A timeout event therefore always coincides with the time-slicing of a VM. However, control is still returned to the previously executing thread when the VM is scheduled again, since the thread did not relinquish control voluntarily.

- **EDVM client to EDVM server**

  In the previous section the effects of an event that is raised while a thread is executing, were discussed. However, when an event is raised while an event handler is executing, it is processed differently. Because an event handler must execute to completion, control cannot be transferred immediately to the handler for the newly raised event. The event is instead queued within the kernel until the event handler has completed and only then is the new event dispatched to the relevant handler. This mechanism ensures that event handlers are not interrupted by other events and therefore prevents global data structures from being updated concurrently.

  However, it is not always necessary to require an event handler to execute till completion. In certain circumstances it is furthermore unacceptable to wait until a handler that is busy executing has finished before the next event can be handled, for example when a high priority interrupt event must be handled quickly. For this reason the rule that an event handler must always execute to completion was relaxed to allow handlers to be interrupted by events that are handled in a different EDVM than that of the handler that was executing when the event was raised. There are no concurrency problems because the two event handlers are in different protection domains and do not share any data structures. After the second event handler has completed executing, the first

handler is resumed. Recursive events are prevented by masking all interrupts that have the same or lower priority than that of the interrupt event that is being handled. This ensures that only higher priority interrupt events can be raised while an interrupt event handler is executing.

- **Event VM client to Thread VM server**

  Event driven VMs would be very restrictive if they could only communicate with other EDVMs. For this reason normal IPC calls to thread VMs are allowed from within an event handler. However, the handler is then blocked and all events that are directed to the handler are queued until the handler has finished executing.

Event Scheduling

In the previous sections it has been shown how the source and target of an event determine the actions that must be performed on that event. This section will examine all the factors that influence event processing and use them to derive an algorithm to determine how an event should be scheduled.

Factors that influence event processing:

- The **Type of event** that has been raised:
  In Section 3.2.3, the different event classes, namely Interrupt, Signal, Transaction and Timeout events are described. Of these, Interrupt and Signal events are handled in the same way and can be grouped together. The valid values for *Type of event* are then *Interrupt*, *Transaction* and *Timeout*.

- The **Relationship** between the source and the target:
  This specifies the relationship between the code that was executing when an event was raised and the handler for that event. Valid values for this relationship are:

  - **None** - The system was idle, meaning that no thread in a VM or event handler in an EDVM was executing when the event was raised.
  - **Same EDVM** - The event was raised while an event handler was executing and the event must be dispatched to an event handler in that same EDVM.
  - **Different EDVM** - The event was raised while an event handler was executing and must be dispatched to a handler in a different EDVM.
  - **Thread** - A thread in a VM was executing when the event was raised.

- **Availability of the Event Handler**:
  Is the event handler available to process the event? Valid values for this condition are *TRUE* and *FALSE*.

The conjunction of these three conditions leads to 24 cases which are described using Dijkstra's guarded command notation [13]:

if Transaction $\wedge$ Different EDVM $\wedge$ FALSE $\rightarrow$ S_0
$[]$ Transaction $\wedge$ Different EDVM $\wedge$ TRUE $\rightarrow$ S_1
$[]$ Transaction $\wedge$ Same EDVM $\wedge$ FALSE $\rightarrow$ S_2
$[]$ Transaction $\wedge$ Same EDVM $\wedge$ TRUE $\rightarrow$ S_3
$[]$ Transaction $\wedge$ Thread $\wedge$ FALSE $\rightarrow$ S_4
$[]$ Transaction $\wedge$ Thread $\wedge$ TRUE $\rightarrow$ S_5
$[]$ Transaction $\wedge$ None $\wedge$ FALSE $\rightarrow$ S_6
$[]$ Transaction $\wedge$ None $\wedge$ TRUE $\rightarrow$ S_7
$[]$ Interrupt $\wedge$ Different EDVM $\wedge$ FALSE $\rightarrow$ S_8
$[]$ Interrupt $\wedge$ Different EDVM $\wedge$ TRUE $\rightarrow$ S_9
$[]$ Interrupt $\wedge$ Same EDVM $\wedge$ FALSE $\rightarrow$ S_{10}
$[]$ Interrupt $\wedge$ Same EDVM $\wedge$ TRUE $\rightarrow$ S_{11}
$[]$ Interrupt $\wedge$ Thread $\wedge$ FALSE $\rightarrow$ S_{12}
$[]$ Interrupt $\wedge$ Thread $\wedge$ TRUE $\rightarrow$ S_{13}
$[]$ Interrupt $\wedge$ None $\wedge$ FALSE $\rightarrow$ S_{14}
$[]$ Interrupt $\wedge$ None $\wedge$ TRUE $\rightarrow$ S_{15}
$[]$ Timeout $\wedge$ Different EDVM $\wedge$ FALSE $\rightarrow$ S_{16}
$[]$ Timeout $\wedge$ Different EDVM $\wedge$ TRUE $\rightarrow$ S_{17}
$[]$ Timeout $\wedge$ Same EDVM $\wedge$ FALSE $\rightarrow$ S_{18}
$[]$ Timeout $\wedge$ Same EDVM $\wedge$ TRUE $\rightarrow$ S_{19}
$[]$ Timeout $\wedge$ Thread $\wedge$ FALSE $\rightarrow$ S_{20}
$[]$ Timeout $\wedge$ Thread $\wedge$ TRUE $\rightarrow$ S_{21}
$[]$ Timeout $\wedge$ None $\wedge$ FALSE $\rightarrow$ S_{22}
$[]$ Timeout $\wedge$ None $\wedge$ TRUE $\rightarrow$ S_{23}
fi

Each S_i represents the action that must be taken when the corresponding guard is true. Some of the guards denote the same state, even though they differ syntactically and therefore the specification can be simplified by merging these guards. The specification can also be

simplified by removing guards that will never be satisfied, due to conditions that are mutually exclusive.

if `Transaction` $\wedge$ `None` $\wedge$ `FALSE` $\rightarrow$ S_6
$[]$ `Transaction` $\wedge$ `None` $\wedge$ `TRUE` $\rightarrow$ S_7
fi

In both of the above guards the *Relationship* state is equal to *None* which means that the system is idle, while the *Type of event* state is equal to *Transaction* which means that the event was raised by a thread in a VM or an event handler in an EDVM. The situation described by the above guards will therefore never occur, since a *Transaction* event is a user event that cannot be raised if the system is idle. The guards can therefore be removed.

if `Transaction` $\wedge$ `Different EDVM` $\wedge$ `FALSE` $\rightarrow$ S_0
$[]$ `Transaction` $\wedge$ `Same EDVM` $\wedge$ `FALSE` $\rightarrow$ S_2
$[]$ `Transaction` $\wedge$ `Thread` $\wedge$ `FALSE` $\rightarrow$ S_4
$[]$ `Transaction` $\wedge$ `None` $\wedge$ `FALSE` $\rightarrow$ S_6
$[]$ `Interrupt` $\wedge$ `Different EDVM` $\wedge$ `FALSE` $\rightarrow$ S_8
$[]$ `Interrupt` $\wedge$ `Same EDVM` $\wedge$ `FALSE` $\rightarrow$ S_{10}
$[]$ `Interrupt` $\wedge$ `Thread` $\wedge$ `FALSE` $\rightarrow$ S_{12}
$[]$ `Interrupt` $\wedge$ `None` $\wedge$ `FALSE` $\rightarrow$ S_{14}
$[]$ `Timeout` $\wedge$ `Different EDVM` $\wedge$ `FALSE` $\rightarrow$ S_{16}
$[]$ `Timeout` $\wedge$ `Same EDVM` $\wedge$ `FALSE` $\rightarrow$ S_{18}
$[]$ `Timeout` $\wedge$ `Thread` $\wedge$ `FALSE` $\rightarrow$ S_{20}
$[]$ `Timeout` $\wedge$ `None` $\wedge$ `FALSE` $\rightarrow$ S_{22}
fi

In all of the above guards the value of the *Availability of the Event Handler* condition is *FALSE*. This means that the event handler is busy and is unable to handle another event. While an interrupt event handler is busy, all interrupt events that are relevant to that handler are discarded. This is done because in most cases it does not make sense to try to queue interrupt events, because a second interrupt will not be raised before the first interrupt has been acknowledged. Another reason is that a device raises an interrupt to signal a certain condition, for example that a byte of serial data has been received. However, if the serial data is not read from the device register before the next byte arrives, the first byte is overwritten. Therefore it is better to discard the first interrupt than to attempt to perform processing of

invalid data. The guards S_8, S_{10}, S_{12} and S_{14} are then reduced to:
if Interrupt $\wedge$ FALSE $\rightarrow S_8$ **fi**

Transaction and *Timeout* events are handled in the same way, irrespective of their source, if their respective event handlers are not available. In each case an event is allocated and initialised with the relevant data and then placed into a queue, until the handler becomes available. The guards can therefore be replaced by the following:

if Transaction $\wedge$ FALSE $\rightarrow S_0$
$[]$ Interrupt $\wedge$ FALSE $\rightarrow S_8$
$[]$ Timeout $\wedge$ FALSE $\rightarrow S_{16}$
fi

The original specification has now been reduced to:

if Transaction $\wedge$ $\wedge$ FALSE $\rightarrow S_0$
$[]$ Transaction $\wedge$ Different EDVM $\wedge$ TRUE $\rightarrow S_1$
$[]$ Transaction $\wedge$ Same EDVM $\wedge$ TRUE $\rightarrow S_3$
$[]$ Transaction $\wedge$ Thread $\wedge$ TRUE $\rightarrow S_5$
$[]$ Interrupt $\wedge$ $\wedge$ FALSE $\rightarrow S_8$
$[]$ Interrupt $\wedge$ Different EDVM $\wedge$ TRUE $\rightarrow S_9$
$[]$ Interrupt $\wedge$ Same EDVM $\wedge$ TRUE $\rightarrow S_{11}$
$[]$ Interrupt $\wedge$ Thread $\wedge$ TRUE $\rightarrow S_{13}$
$[]$ Interrupt $\wedge$ None $\wedge$ TRUE $\rightarrow S_{15}$
$[]$ Timeout $\wedge$ $\wedge$ FALSE $\rightarrow S_{16}$
$[]$ Timeout $\wedge$ Different EDVM $\wedge$ TRUE $\rightarrow S_{17}$
$[]$ Timeout $\wedge$ Same EDVM $\wedge$ TRUE $\rightarrow S_{19}$
$[]$ Timeout $\wedge$ Thread $\wedge$ TRUE $\rightarrow S_{21}$
$[]$ Timeout $\wedge$ None $\wedge$ TRUE $\rightarrow S_{23}$
fi

Some of these guards specify the same action and can be grouped together.

- The guards of S_1 and S_3 denote the same action. In each case an EDVM was executing when a *Transaction* event was raised. It does not make a difference to the event scheduling whether the event should be handled in the same EDVM, or a different EDVM from where it originated.

- The guards of S_{17} and S_{19} also denote the same action. As is the case with *Transaction* events, it does not make a difference to which EDVM the *Timeout* event must be dispatched. In each case a *Timeout* event is allocated and placed in the relevant event queue.

Of the original 24 actions that were specified, only 12 are unique. The specification is therefore reduced to:

if `Transaction` $\wedge$ $\wedge$ `FALSE` $\rightarrow S_0$
$\square$ `Transaction` $\wedge$ `All EDVMs` $\wedge$ `TRUE` $\rightarrow S_1$
$\square$ `Transaction` $\wedge$ `Thread` $\wedge$ `TRUE` $\rightarrow S_5$
$\square$ `Interrupt` $\wedge$ $\wedge$ `FALSE` $\rightarrow S_8$
$\square$ `Interrupt` $\wedge$ `Different EDVM` $\wedge$ `TRUE` $\rightarrow S_9$
$\square$ `Interrupt` $\wedge$ `Same EDVM` $\wedge$ `TRUE` $\rightarrow S_{11}$
$\square$ `Interrupt` $\wedge$ `Thread` $\wedge$ `TRUE` $\rightarrow S_{13}$
$\square$ `Interrupt` $\wedge$ `None` $\wedge$ `TRUE` $\rightarrow S_{15}$
$\square$ `Timeout` $\wedge$ $\wedge$ `FALSE` $\rightarrow S_{16}$
$\square$ `Timeout` $\wedge$ `All EDVMs` $\wedge$ `TRUE` $\rightarrow S_{17}$
$\square$ `Timeout` $\wedge$ `Thread` $\wedge$ `TRUE` $\rightarrow S_{21}$
$\square$ `Timeout` $\wedge$ `None` $\wedge$ `TRUE` $\rightarrow S_{23}$
fi

Each of these actions has been implemented in an algorithm that performs the necessary event processing. The following code shows a simplified version of this algorithm.

```
PROCEDURE ScheduleEvent;
VAR
   eventType: EventType;          (* the type of event that was raised *)
   event: Event;                  (* event structure that describes an event *)
   targetHandler: EventHandler;
BEGIN
   eventType := GetEventType();
   (* allocates storage space for the event *)
   event := AllocateEvent(eventType);
   IF eventType = TransactionEvent THEN (* S0, S1, S5 *)
      (* perform transaction event scheduling *)
   ELSIF eventType = InterruptEvent THEN (* S8, S9, S11, S13, S15 *)
     (* determine the handler to which the event must be dispatched *)
     targetHandler := GetTargetHandler(event);
     IF targetHandler = NIL THEN (* S8 *)
        DiscardEvent(event) (* no handler is available *)
     ELSE
        (* initialise the event with the relevant parameters *)
        InitialiseEvent(event);
        IF NothingWasExecuting() THEN (* S15 *)
           (* neither a thread in a VM nor an EDVM was
              executing when the interrupt event was raised *)
        ELSIF ThreadWasExecuting() THEN (* S13 *)
           (* Store the executing thread first *)
           StoreThread();
           (* place the event at the back of the event queue *)
           EnterEvent(eventQ, event)
        ELSE (* S9, S11 *)
           IF targetHandler # GetSourceHandler() THEN (* S9 *)
             (* The event is handled in a different EDVM than the executing
                EDVM. Place the event at the front of the event queue
                so that it will be handled immediately. *)
             PushEvent(eventQ, event)
           ELSE (* S11, event is handled in the same EDVM *)
             EnterEvent(eventQ, event)
           END
        END
      END
   ELSE (* S16, S17, S21, S23 *)
     (* perform timeout event scheduling *)
   END;
   SwitchToHandler(targetHandler, GetEvent(eventQ));
END ScheduleEvent;
```

3.3 Implementation

The code to handle Event Driven VMs can be broken down into two distinct pieces of code: the code within the kernel that creates, schedules and dispatches events, and the user-level code within an EDVM that provides library calls to the user. The code within the kernel is more complex and interesting than the user-level code and the following sections will therefore concentrate on this code. The implementation will be examined by looking at how each type of event is handled.

3.3.1 The Interrupt Event Handler

After an interrupt event is raised, it is first dispatched to a generic interrupt handler within the kernel. The generic handler performs the processing that is common to each interrupt event and dispatches the event to the correct event handler. The most important part of this processing consists of determining whether a thread or an event handler was executing when the event was raised and then performing the appropriate processing.

3.3.2 The Transaction Event Handler

The primary way in which data is transferred between a client thread and an EDVM is via page re-mapping. Page re-mapping is an excellent way to transfer large amounts of data across protection boundaries, since the data is not really copied. Instead the page tables are manipulated in such a way that the data buffers are added to the address space of the target EDVM. However, if the data buffers are small it is more efficient to copy the data than to manipulate the page tables. Therefore a distinction is made between a message passing operation that uses page re-mapping to transfer large amounts of data and a message passing operation that only uses a small amount of data. In the rest of this thesis, a *RemapEvent* will refer to a transaction event that makes use of page re-mapping, while a *CopyEvent* will refer to a transaction event that makes use of in-memory copying. In the Gneiss kernel, short messages are defined as being of length 64 bytes or less. The necessary processing to distinguish between page re-mapping and copying is hidden within the kernel and the user has no need to know which mechanism was used.

When a server thread within a VM executes a *ReceiveRequest* operation, a pointer to a buffer in user space is provided. When an IPC message is received, the message data is then copied to this buffer. However, in the case of EDVMs the kernel does not have access to a buffer

in user space, since an event is dispatched by the kernel. This means that the kernel must allocate message buffers to store the message data.

Before a *RemapEvent* can be raised, the user-level client must first use a kernel call to allocate memory that can be used for page re-mapping. A small buffer is reserved within this re-mappable memory to hold the event data structure. When the event is raised, the kernel fills the fields of this structure with information that identifies the event uniquely, as well as with pointers to the data being transferred. A *CopyEvent* makes use of buffers that are allocated in user space and are managed by the kernel. When a *Transaction* event handler is installed, a pointer to these buffers is passed to the kernel. This buffer is then managed by the kernel and used as a target buffer for small messages.

There are two reasons why this scheme is used instead of copying the event data to the stack of the event handler:

1. If the event structure and its associated data are copied to the stack of the event handler, it cannot exist independently of that handler, except if the data is copied *again* to some global data structures. This means that the handler is busy until the event has been processed to completion, which could take a long time if a lengthy request is being processed. During that time, the event handler cannot accept any other requests.

2. The stack of the event handler is in a different address space from the client thread. The event structure must therefore be built in a temporary buffer within the kernel before it can be copied to the event handler, since data cannot be copied directly from the client thread to the event handler.

3.3.3 The Timeout Event Handler

It is important that EDVMs provide support for timeouts that can be set to expire after a specified interval. Such timeouts are used to terminate a user request if a peripheral device does not respond after a specified time. Timeouts are also used to perform device specific operations, like stopping the motor of a diskette drive if no requests were received for a certain time.

In the Gneiss kernel, timeouts are implemented as a linked list that is sorted according to the time that is left before the timeout expires. Each item in the list contains the number of timer ticks before the next timeout must occur. The aim of this mechanism is to simulate multiple timers by using only one timer as described by Tanenbaum [36]. The same mechanism is

used in an EDVM to simulate multiple timers while only a single kernel timer is used. Each EDVM maintains a list in user-space of all the timeouts that are set for that EDVM. The advantage of this mechanism over using a kernel timer for each timer in the EDVM is that it is only necessary to perform a kernel call when an operation is done on the first timeout in the list, for example when the first timeout is removed or a new timeout is added to the front of the list.

When a kernel timer expires, a generic handler is called within the kernel. A timeout event is then created and the event is dispatched to the relevant handler. Thereafter the kernel timer is set to the interval of the next item in the list of timeouts. However, this list is in user space and cannot be reached and traversed easily from the kernel. There are a number of ways to handle this problem:

1. Perform a kernel call when a timeout operation has an impact on the first *and second* items in the timeout list, instead of just the first. The kernel then has access to the interval of the next timeout in the timeout list after the first timeout has expired.
2. The user-level timeout handler could return the offset of the next timeout to the kernel. This value is then used to reset the timer.
3. The value of the second timeout is kept in a variable that is shared by the kernel and the EDVM. The value of this variable is kept consistent with the actual contents of the list.
4. Move the list of waiting timeouts to the kernel. Kernel calls could then be used to set and cancel timeouts.

Option two seems to be the most elegant option, but suffers from a critical flaw. The next timeout is only reset after the user-level timeout handler has finished executing. It will therefore be incorrect, since no timer was active while the user-level handler was executing. It is possible to compensate for this by determining the length of time that the user-level handler executed, but at the cost of increased complexity.

Option one suffers from an extra kernel call in comparison with options two and three, while option three uses shared memory between the kernel and the EDVM. The easiest option to implement is option four, since it uses only the existing mechanism within the kernel. However, this means that a kernel call must be performed each time a timeout is set or cancelled which is inefficient. Option three has been selected, because it involves fewer kernel calls than option one and the kernel can read data from user space in a safe way. Even if

the EDVM is corrupted and the shared variable contains an incorrect value, the only impact would be on the corrupted VM. Figure 14 shows how the list of timers works.

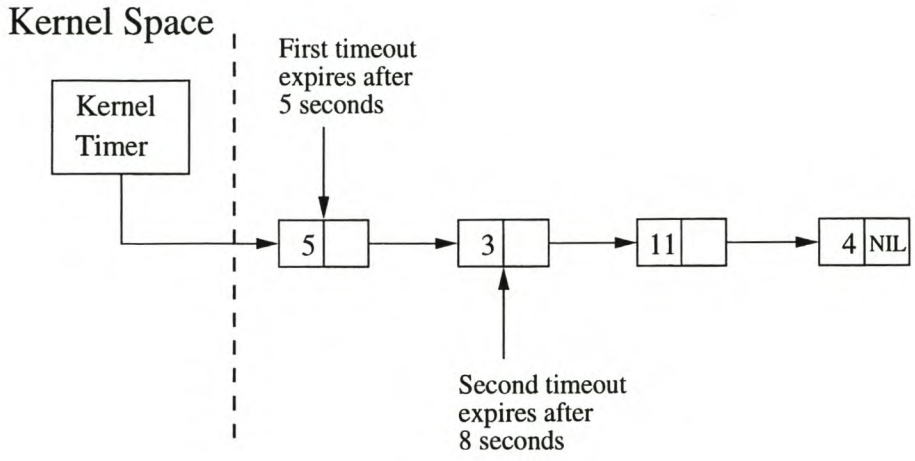

Figure 14: Timers in an EDVM

3.3.4 Efficiency

The greatest disadvantage of user-level device drivers is that they are less efficient than in-kernel device drivers. This lack of efficiency must be minimized to make user-level device drivers a viable option. The following factors influence the efficiency of user-level device drivers that are implemented within an EDVM:

1. The cost of the extra context switches from the kernel to the user-level driver and back to the kernel.
2. The cost of message passing between the client and the EDVM.
3. The time that is spent in the kernel to identify and determine the availability of the event handler.
4. The time that is spent doing event processing depending on whether the event was raised by a hardware source, a user thread or an event handler.

3.3.5 Improving IPC performance

IPC performance is critical for the performance of user-level device drivers as well as the micro-kernel approach as a whole. Chen and Bershad compared applications under Ultrix and Mach on a DECStation 5200/200 and found that, in some cases, Mach was up to 66% slower than Ultrix [7]. IPC overhead was cited as the primary factor that contributed to Mach's poor performance.

Due to the performance penalty of IPC, critical device drivers and servers were integrated within some micro-kernels. Mach [9] and Chorus [19] are examples of systems that use this approach. However, it has been shown that a cross-address space round-trip IPC operation, from user-level thread to user-level thread, can be performed in as little as $10\mu s$ on the i486 architecture [27]. This is comparable to the duration of a kernel call in a typical monolithic kernel.

Since the Gneiss kernel is part of the address space of each VM, an in-kernel device driver can access the address space of the VM from which it was called. Even more, by manipulating the page directory, an in-kernel device driver can access the address space of *any* VM in the system. This makes it easy to copy data efficiently between the kernel and a user-level client.

When a device driver is implemented in an EDVM, data must be copied between two distinct protection domains. Unfortunately, the i386/486 architecture supports only one active address space at a time. It is therefore not possible to copy data directly between a user-level device driver and a client buffer. One solution is to use a temporary buffer in the kernel as an intermediary between the two address spaces. However, two copy operations are then needed to transfer a buffer between a client thread and an EDVM. This is inefficient and the kernel must also allocate and manage a number of temporary buffers which add further overhead. This means that the overhead associated with this method is unacceptably high and it is unsuitable for all but the shortest copy operations.

Another option is to use a data area that is permanently shared across all protection domains. All VMs and EDVMs have read-write access to this predefined area and can deposit data there. A server can retrieve data from this area and process it in its own protected address space or process it directly. The DEC Firefly RPC facility is an example of this approach [35]. This technique reduces the number of data copies, but does not eliminate it completely. It also has serious security risks, since the data area is accessible to all VMs which allows a malfunctioning or malicious thread to corrupt the buffers of other threads. This technique is therefore not suitable.

There are various virtual memory techniques that can be used to transfer data between different protection domains. These techniques all depend on the ability of the Memory Management Unit (MMU) to handle synonyms. The term synonym describes different virtual addresses that translate to the same physical address. By manipulating the virtual addresses it is possible to transfer data buffers between different protection domains without any in-memory copying.

Copy-on-write (COW) is implemented by modifying the page tables of the server to create a read-only mapping to a client message buffer. A server can then read data from the client message buffer without having to copy the data. When the server performs a write operation on the read-only data buffer, a page fault occurs. The page fault is handled by the kernel and the message buffer is then copied to a buffer that is supplied by the server. Copy-on-write can be inefficient when the server often modifies the client buffers, since page faults are expensive to process.

Page re-mapping is analogous to a move operation. The pages of the client buffer are re-mapped into the address space of the server. The client and the server now share this buffer. The buffer is therefore a critical section that must be protected from concurrent access. This is enforced by modifying the page table entries that describe the client buffer to allow read-only access alone. These entries are changed to read-write access when the IPC operation has been completed.

An important constraint of page re-mapping is that the size of the data buffers must be a multiple of 4K, since that is the page size on the ix86 architecture. Therefore memory for a message buffer in a client thread must be allocated on a 4K boundary and the size of the message buffer must also be a multiple of 4K. Since any piece of dynamically allocated memory is not guaranteed to adhere to these restrictions, a client thread must prepare its message buffers beforehand. In practice, this does not pose a significant problem, since buffers can be pre-allocated using a kernel call that is provided for this purpose.

A buffer that does not adhere to these restrictions could cross a page boundary or occupy only part of a page. In the first case, the buffer would be incompletely mapped. In the second case, a part of the client process's memory that is not part of the buffer will also be visible to the EDVM. It would then be possible for the EDVM to modify this memory. Both cases are unacceptable and restrictions are therefore enforced by the kernel.

A page table is used for each message buffer. This wastes $4Mbyte - x$ of address space where x is the size of the buffer. This also means that a client VM can allocate a maximum of

256 message buffers, since only 1 Gigabyte of address space area is reserved for this purpose. However, since message buffers can be re-used, this does not pose a significant problem and since up to 4 Megabytes can be addressed with a single page table, the maximum size of a message buffer is also 4 Megabytes.

The advantage of the above approach is that only the relevant page directory entry needs to be changed when a message buffer is re-mapped. There is no need to copy the entry for each page or to allocate a new page table if all the pages in the previous table have been used. For example, if the message buffer has a size of 4 Megabytes, the relevant entries for each of the 1024 pages would have to be modified during a re-map operation instead of modifying only a single page table entry. Therefore it is more efficient to use a complete page table for each message even if some addressing space is wasted.

3.3.6 Debugging and Exception Handling

User-level device drivers execute as user programs, which means that all debugging tools used for debugging user applications can also be used to debug device drivers. When a critical error occurs, only the EDVM that contains the driver is terminated, while the kernel and the rest of the system remain intact. It is possible to identify the error and test the driver again without the need to recompile or reboot the kernel. This reduces development time significantly.

3.3.7 Disabling the EDVM

When a debug breakpoint is reached within an event handler, the EDVM must be stopped. In the case of a thread in a VM, this is achieved by refraining from scheduling the VM. However, an EDVM is scheduled by events that are raised asynchronously. It is therefore possible for other event handlers in the EDVM to be invoked while the first handler is stopped by the debugger. This would allow concurrent access to shared data structures which must be prevented.

After an event handler has been stopped for debugging, all the other event handlers in the EDVM are identified by the kernel and disabled. In the case of hardware interrupt event handlers, this also entails masking the interrupt that the handler is servicing. If a timeout event was set, the timeout is stopped, but the amount of time that was left before the event would have been raised is saved.

The user can then examine the driver and debug it at his leisure. When the driver is restarted, either under control of the debugger or for normal execution, the event handlers are re-enabled. The timeouts are also reset, not to their original length, but to the amount of time that was left when they were unloaded. This ensures that the debugging operation is transparent to the driver that is being debugged.

3.3.8 A Fatal Exception within an EDVM

When a fatal exception occurs while an event handler is executing, the whole EDVM is destroyed. This is in contrast with a thread-based VM where only the offending thread is destroyed. Within an EDVM, the loss of an event handler is a critical error since each handler is unique. A VM that makes use of a number of duplicate threads could continue even if one is destroyed, although in practice the other threads also eventually cause an exception due to the same error.

3.4 Conclusion

The non-deterministic, call-back nature of the event model is more convenient than the thread model for the implementation of non-deterministic applications like device drivers. However, a disadvantage of the event model is that the kernel must allocate storage space for the data of each event that is dispatched to a handler. In comparison, a thread in a VM can supply a buffer on the stack as part of a kernel call and this buffer can be used as a target for message data.

Events can be either blocking or non-blocking and the thread model can also support synchronous or asynchronous message passing. The synchronization and mutual exclusion primitives that are supplied are also implementation specific and do not depend on a specific model. For example, conditional critical regions can be used in the thread as well as the event models.

The conclusion is that the thread model and the event model are equivalent in terms of functionality for implementing device drivers. This can be illustrated by the fact that each model can be used to emulate the other. The event model is somewhat more convenient in some areas, for example to implement a device driver, while the thread model is easier to use in others, for example to implement a file server. However, how well a specific model is suited to implement device drivers is not determined primarily by whether it is the event model or thread model, but by the synchronization and message passing primitives that are

supplied. The primitives are implementation specific and differ from operating system to operating system.

The following functionality is necessary to implement device drivers in user space:

- It must be possible to handle interrupts.
- It must be possible to handle more than one request at a time to allow shareable device drivers to be implemented.
- There must be some synchronization and mutual exclusion primitives to control access to the device controller.

This chapter has shown how event driven virtual machines (EDVMs) can be used as a stable and productive environment for device driver development in user space. The design and scheduling policies concerning EDVMs have been discussed and methods for improving the efficiency of EDVMs have been outlined. Lastly, it has been shown why it is easier to debug user-level device drivers than in-kernel drivers.

Chapter 4

Evaluation

A number of device drivers have been developed to compare the strategies described in this thesis with respect to 1) the efficiency of the implemented driver and 2) the extent to which the development strategy facilitated the implementation process. Emphasis is placed on the concepts that cause one strategy to be more efficient than the other. This is done by examining and discussing execution paths instead of implementation specific timing results. Since it is difficult to quantify ease of development, no measurements will be given, but instead an argument will be presented to highlight the merits and drawbacks of the various development strategies.

4.1 How to Measure Performance

Two different methods of measuring performance have been considered. In the first method, a hardware monitor is used to perform an analytical measurement by identifying the instructions that are part of the execution path of the test run and then computing the total execution time. This method gives exact timing results but a dedicated hardware monitor is needed. Certain processors, like the Intel Pentium [23] and Pentium Pro [22] processors, also have additional instructions that can be used to gather statistics about code execution.

The second method consists of determining the execution time experimentally by inserting timing code in the test program. The relevant operation is then executed n times and after compensating for overhead due to the measurement code, the elapsed time can be used to compute the average duration of the operation. This method gives acceptable results if n is large enough.

The second method was selected, because a hardware monitor was not available and because performance measurement with timing code yielded acceptable results for comparing the various device driver development strategies.

4.1.1 Determining the efficiency of Event Driven VMs

The first step was to determine how much overhead is caused by the basic operations of the Gneiss kernel. The Gneiss micro-kernel has been implemented in the Oberon language. Oberon is a high level language which means that the code generated by the Oberon compiler is generally not as efficient as hand-crafted assembler code. The Gneiss micro-kernel has also been designed as an educational kernel, which means that the most important design constraint was that the kernel must be as simple and generic as possible. However, since a cross-address space round-trip IPC operation takes $140\mu s$ on a 486DX2/66, the speed of the kernel still compares favourably with other micro-kernels like Mach and Amoeba [14, 18].

In the following sections, a breakdown of the execution path of each device driver development strategy will be examined. The performance overhead of each portion of the execution path will be discussed, as well as whether the overhead could have been eliminated by a different design.

4.1.2 Control Transfer to the Kernel

The only way in which control can be transferred to the kernel is via an interrupt, a kernel call or an exception. However, since exceptions and kernel calls are just specialized interrupts, only the cost of an interrupt needs to be examined. The basic cost of a control transfer to the kernel and a transfer to user space always forms part of any measurement of the performance of the various development strategies. The code excerpt in Figure 15 shows how the interrupts are processed in the Gneiss kernel.

This discussion assumes that an interrupt was triggered while a thread was executing. The Gneiss kernel is part of each VM's address space, but is protected from user access by the paging mechanism[1] of the ix86 processor. When an interrupt occurs, control is transferred through an interrupt gate to the kernel [11, 22, 23]. As part of the control transfer, a stack switch to a ring 0 stack is also performed. The necessary stacks for ring 3 and ring 0 must have been set up beforehand in the Task State Segment (TSS), but it is the processor that

[1]The kernel executes at ring 0 while user-level VMs execute at ring 3

```
User-Space code is executing
   Interrupt is triggered
   Control Transfer to kernel
   Store state of executing thread in register storage area
   Set up stack frame for high level generic interrupt procedure
   Call interrupt procedure

   PROCEDURE FieldInterrupt(intNum: SHORTINT; state: RegisterStatePtr);
   VAR
      thread: Thread;
   BEGIN
      (* call specific interrupt handler *)
      intTable[intNum].handler(intNum, state);
      IF IsHardwareInterrupt(intNum) THEN
         AcknowledgeInt(intNum)
      END;
      IF ShouldReschedule() = FALSE THEN
         ReturnFromInterrupt(state)
      ELSE
         (* Call the scheduler to select a new thread to execute *)
         SelectThread(thread);
         (* wait for an interrupt if there are no threads that are ready
            to execute *)
         IF thread = NIL THEN WaitForInterrupt() END;
         (* make the address space in which the thread executes active *)
         SetPageDirectory(thread.vm.pd);
         (* Switch to user-level *)
         TransferTo(thread)
      END
   END
```

Figure 15: Low-level interrupt processing

performs the stack switch as part of the control transfer. The processor also stores the necessary information, like the user-level stack segment and stack pointer, flags and user-level instruction segment and instruction pointer on the ring 0 stack so that an interrupt return (IRET) instruction can later return control to the user-level. This information, along with the contents of the general registers of the executing thread, must be stored until the thread is again scheduled to execute. Unfortunately the ix86 processor does not allow an interrupt handler to be called like a procedure in a high level language. Instead, control always jumps to the handler. The interrupt handler must therefore be written in assembly language (except in the case where a high level language makes special provision for interrupt handlers). The Oberon compiler expects the stack to be set up in a certain way before a procedure is called. A stack frame must therefore be set up manually before the first high level procedure can be called.

This procedure is shown in Figure 15 and it is used to field all interrupts, exceptions and

kernel calls and to vector them to their respective handlers. After a hardware interrupt has been handled to completion, the interrupt controller is acknowledged. The scheduler is then invoked with the *SelectThread* procedure to select the next thread that is ready to execute. If a ready thread was found, a context switch is performed from kernel space to user space and the selected thread starts executing. However, if no threads are ready for execution, interrupts are enabled and the kernel waits until an interrupt occurs. This is more efficient than to schedule a generic null thread because control stays within the kernel instead of in user space and the kernel can therefore respond more quickly to the next interrupt or exception.

4.1.3 In-Kernel Device Driver Performance

In this section, the different sources of overhead when an interrupt is handled by an in-kernel device driver will be examined in more detail. It is assumed that a user-level thread was executing when the interrupt occurred. The sources of overhead are:

1. **Context switch**
   In Section 4.1.2 it is shown that the basic cost of a context switch from user-space to the kernel and back is determined to a great extent by the hardware of the processor. For example, an Intel 486 DX-50 processor takes about 2 micro-seconds to enter and leave the kernel if the kernel is part of the executing task but at a different privilege level.

2. **Storing the thread state**
   When an interrupt occurs, the registers of the executing thread must be stored so that the thread can later be resumed from the point where it was interrupted. On the i386, all of the general registers can be saved on the stack with a single instruction that takes only 11 CPU cycles. However, it is not always necessary to save all the registers. A carefully coded interrupt handler could make use of only some of the registers, thereby making it unnecessary to save the unused registers. However, this places an additional burden on the programmer to save and restore all the registers that are used. This is cumbersome when assembly language is used and very difficult to do when a high level language is used.

3. **Setting up the stack frame**
   The process of setting up the stack frame consists of only a couple of instructions to

prepare the stack for the first high level procedure call. This overhead is eliminated if the kernel is implemented in a low-level language such as assembly.

4. **The device specific interrupt handler**
   In the Gneiss kernel, control passes through a layer of indirection before the specific interrupt handler is called. This makes the interrupt handling more generic and it also makes it easier to code a specific interrupt handler. The extra overhead of this indirection could be avoided by replacing the generic interrupt handler with a number of dedicated handlers, but this overhead is not significant.

   The amount of time that is spent within the interrupt handler is dependent on the device hardware itself, but should of course be as short as possible.

5. **Acknowledging the interrupt**
   On the ix86 hardware, an interrupt must be acknowledged after it has been processed. This entails writing one or two bytes to an I/O port and takes only a couple of cycles of execution time. Systems that have a generic interrupt handler, like the Gneiss kernel, have some extra overhead, since each interrupt must be examined to determine whether it is a hardware interrupt.

6. **Scheduling the next executable thread**
   After an interrupt has been processed, the scheduler is invoked to select the next thread that is ready to execute. In the Gneiss kernel, the only way a thread can relinquish control is by performing a kernel call. According to this model, a thread can be interrupted by a hardware interrupt, but after the interrupt has been handled in the kernel, the thread must immediately be resumed. The scheduler relieves the programmer from the necessity of identifying the interrupted thread and explicitly returning control to that thread.

   This is but one way of handling interrupts. Other kernels support preemptive threads which would make it acceptable for a thread to lose control due to a hardware interrupt or the expiry of a time slice.

   The scheduler has the potential to be a big source of overhead. The Gneiss kernel uses a simple round robin scheduling method with queues for each priority level. If a client and a server in different address spaces pass messages back and forth between each other, the total scheduling overhead is 12% of the total time of a round-trip IPC operation. It should be noted that the client and the server perform no processing except to pass messages back and forth and that the scheduler is invoked after each kernel call. This means that the scheduler is invoked three times during a round-trip IPC operation since

the operation consists of a *Transaction*, *ReceiveRequest* and *SendReply* combination. If another scheduling algorithm is used, the overhead would be different.

7. **Restoring the state of selected thread**
   After a thread has been selected to execute, the contents of the processor registers are restored to their state at the time when the thread was interrupted. This is the counterpart of step 2 (above) and everything that is done in that step must be done in reverse order.

8. **Return to user-level**
   Control is returned to the user-level by means of performing an interrupt return instruction. As is the case with the *Control transfer to the kernel* step, the overhead is primarily hardware dependent.

An important point that is highlighted by a breakdown of the overhead is that a generic implementation is usually less efficient than a specific implementation. For example, in the Gneiss kernel, interrupt handling consists of two levels of indirection. The real interrupt handler is a piece of assembly code that calls a generic high level interrupt handler called *FieldInterrupt*. This handler performs generic processing and vectors the interrupt to a specific interrupt handler. This mechanism is less efficient than to implement a handler in assembly code and to handle each interrupt individually. However, the advantage of the generic handler is that a kernel-resident device driver can use a normal procedure as an interrupt handler which means that the developer does not need to be concerned about saving and restoring the registers, acknowledging the interrupt or calling the scheduler.

Now that we have looked at the overhead of an interrupt handler within the kernel, the next step is to examine the overhead that is associated with a user request to an in-kernel device driver. In the Gneiss kernel, device drivers are implemented as kernel resident servers (see Section 2.3.5) and the *Transaction* primitive is used to communicate with these servers. The kernel must determine whether the target of the *Transaction* primitive is a kernel resident server or another user-level thread. This is not an expensive operation and takes the form of a number of comparisons. A small amount of time is also spent in passing parameters to the request handler within the kernel, but since these parameters are pointers to data in user space, the overhead is not excessive.

4.2 User-Level Interrupt Event Performance

In the previous section, the performance of in-kernel drivers was examined with special emphasis on their interrupt and user request handlers. This provides a benchmark with which to gauge the overhead that is incurred when device drivers are implemented as user-level drivers. The benchmark is valuable because user-level device drivers suffer from most of the overhead of in-kernel drivers along with some additional overhead. The purpose of this section is to determine whether this additional overhead is significant enough to make user-level device drivers infeasible. This discussion assumes that a thread was executing when an interrupt event was raised and that the event was dispatched to a user-level event handler.

1. **Transfer to kernel**
   Items 1-3 of Section 4.1.3, namely the transfer of control to the kernel, the storing of the state of the thread and the setting up of the stack frame, still apply.

2. **Determine the target user-level VM**
   The user-level VM that handles the interrupt must be identified. This can be done with a hash table or a simple array lookup and does not cause significant overhead.

3. **Schedule interrupt event**
   The scheduling of an event was discussed in Section 3.2.10. Scheduling consists of a number of jumps and queue operations. Therefore it is a relatively expensive operation and should be carefully optimized. In the optimal case, the event is dispatched to a handler immediately and the handler executes to completion without any interruption. However, it is possible for a lower priority interrupt event handler to be interrupted by a higher priority interrupt. In that case, the the lower priority handler will execute for a longer time. Fortunately the peripheral devices that raise low priority interrupts are usually slow and a small delay in the interrupt handling can be tolerated.

4. **Switch control to user-level interrupt handler**
   After the target event handler has been identified, a stack frame is prepared and a context switch to user space is performed. The actions that are taken are similar to those described in Items 7 and 8 of Section 4.1.3, namely to restore the state of the event handler and return to user-level via an IRET instruction with the difference being that the transfer is done to an event handler instead of to a thread.

5. **Return to kernel**
   After the event handler has finished executing, control returns to the kernel by means of

a user interrupt. The steps that have been described in Items 1-3 of Section 4.1.3 must then be performed again and the user interrupt is vectored to a generic return-to-kernel handler.

6. **Performing cleanup**
   The return-to-kernel handler identifies the type of event handler that has finished executing and performs the necessary cleanup operations. For example, in the case of a hardware interrupt event handler, the interrupt must be acknowledged. The stack pointer, stack parameters and instruction pointer of the event handler are also restored to their original values so that the handler is ready to handle the next event. The overhead of the cleanup is only a number of cycles, since it consists of output to a port as well as some memory assignments.

7. **Handle next event**
   After the necessary cleanup has been done, the next event can be handled. If there are no more events that are ready to be handled, control is returned to the thread that was originally interrupted.

The above section shows that to handle an interrupt in a user-level driver, a number of operations have to be performed that are not needed for an in-kernel driver. Of these operations the greatest overhead is caused by the scheduling of the interrupt event, the context switch to user-level and the context switch back to the kernel after the user-level interrupt handler has completed.

4.3 User-Level Request Handler Performance

Along with the ability to field interrupts, a user-level driver must also be able to handle user requests. In this section, the overhead of a client request to a user-level device driver will be compared to the overhead of handling a client request within an in-kernel device driver.

1. Steps 1-3 of Section 4.1.3 are first performed before the generic handler that dispatches IPC events to an Event Driven VM is called.

2. **Identify the target EDVM**
   The number of the IPC port that is used to transfer the client request is used to identify the target EDVM. This port number is a 32 bit word and can range from 0 to 2^{32}. A hash table is used to map the port number to the target EDVM. The need for a hash

table can be eliminated if the client specifies the target EDVM directly, for example by providing a pointer to the necessary data structures. However, this information must still be validated in some way to prevent the client from supplying incorrect information that could cause the kernel to malfunction. Naturally the process of validation takes some time. It was also found that the extra overhead when ports are used is minimal (in the order of micro-seconds) and that the extra layer of abstraction is very useful since it de-couples the client and the server.

3. **Resource allocation**
   Since an event is not bound to a specific event handler, the in-kernel data structure that describes an event handler cannot also be used to store event-specific information. Instead, memory must be allocated to store the information that describes each new event. Memory allocation and deallocation is a relatively expensive operation and it is important to reduce this cost in some way. One method is to make use of the fact that all the event data structures are the same size. A number of memory blocks are pre-allocated and linked to form a list. Whenever storage space is needed for a new event, a block is removed from this list and used. When the block is not needed any more, it is returned to the list. The only time when new memory must be allocated is when the free list is empty. This should happen infrequently, but depends on the number of blocks that were pre-allocated and the number of events that must be queued. Memory allocation and deallocation is therefore reduced to removing and adding an element to a list.

4. **Check if the handler is available**
   If the event handler or EDVM is not available, the event is placed in a queue.

5. **Synchronization**
   Because transaction events are blocking, the client that raised a transaction event must be blocked until the event has been handled to completion. If message passing was non-blocking, it would have been unnecessary to store information describing the client. The synchronization overhead is a number of cycles to store the information of the client.

6. **Data transfer**
   The biggest overhead penalty of user-level request handlers in comparison to in-kernel request handlers, is the cost to transfer data between different protection domains. In the kernel, device drivers can access client data directly because the kernel can access any VM's address space. However, to access a different address space than the active address space, two TLB flushes must be performed, which also adds to the overhead. In

some cases it is also better to store data in an intermediate buffer in the kernel before copying it to user space.

A number of techniques to improve IPC performance are described in Section 3.3.5. The best compromise is a combination of an IPC message that is optimized for short messages and an IPC message that is optimized for large messages. In-memory copying of the registers of the processor is typically used for the short messages and page remapping is used for large messages [25].

7. **Preparing the user-level event structure**
   Unfortunately the data structure that is described in the *Resource Allocation* section cannot be accessed from the user-level, since it is part of the address space of the kernel. Therefore, a separate data structure that is accessible from a user-level event handler must be prepared and filled in with the relevant data pointers to describe the event. It also takes some extra overhead to allocate and prepare this structure.

   An alternative is to allocate event structures on page boundaries and to map them read-only into the address space of the EDVM. The same event structure can then be used by the user as well as the kernel. However, the kernel must ensure that the event structure contains no information that could cause security problems and the added complexity of performing page remapping operations makes this mechanism infeasible.

8. **Context switch to user-level**
   A context switch to user-level is performed and the event handler executes till completion. Steps 1-3 of Section 4.1.3 are performed and a generic return-to-kernel handler is called.

9. **Restart client thread if event has not been blocked**
   After the event handler has executed to completion and issued an interrupt, the kernel determines whether the event was blocked. If that is not the case, the client thread that initiated the request must be restarted. The reply to the IPC message is then transferred to the address space of the client thread after which the thread is scheduled for execution. This operation is the counterpart of the *Data Transfer* and it is also expensive.

10. **Prepare event handler for next event**
    The event handler is then prepared to handle the next event by restoring its instruction pointer and stack pointer values.

11. **Handle next event**
    After the necessary cleanup has been done, the next event can be handled. If there are

no more events that are ready to be handled, the scheduler is invoked to select a thread for execution.

To handle client requests in a user-level driver, a number of operations also have to be performed that are not needed for a request handler in an in-kernel driver. Similar to user-level interrupt handlers, an extra context switch to user-space and another switch back to the kernel have to be performed. Another source of overhead is the scheduling of events as well as the overhead that is caused by resource allocation and synchronization. However, the greatest source of extra overhead is due to the cost of transferring data between different protection domains.

The above comparisons show that user-level interrupt handlers and request handlers are inherently less efficient than in-kernel handlers. The crucial question is whether this lack of efficiency is an acceptable trade-off to gain the advantages, like safety and faster development times, that are enjoyed by user-level drivers.

As part of this thesis, a number of user-level device drivers were developed, including a serial driver, an IDE disk driver and a SCSI driver. It was found that only the serial driver suffered some data loss due to higher interrupt latency, and only at baud rates in excess of 57600 baud on a slower computer like an i386. At the other end of the spectrum, the SCSI driver was used to transfer huge amounts of data between a client thread and a user-level driver. It was found that the limiting factor there was not the efficiency of the device driver, but rather the data transfer rate of the peripheral device itself. The above experiments have shown that it is indeed worthwhile to implement device drivers in user-space.

4.4 A Comparison of the Ease of Development of Device Drivers

It is difficult to quantify and measure how easy it is to develop a device driver. One approach is to implement a certain device driver a number of times and to use a different development strategy each time. This makes it possible to measure the length of time that is taken by each implementation and compare it against the time that the other implementations took. Unfortunately this approach does not give accurate results, because when a driver is implemented more than once the implementation process becomes easier for each subsequent time due to the experience that was gained during the previous implementations. This problem can be circumvented if a different developer is assigned to each development method. The drawback of this strategy is that the knowledge and ability of each developer is different, but

if a large enough number of developers are used, the differences between the developers will average out. This is the preferred method of measuring the ease of development, but because of a lack of manpower, this strategy could not be used.

As an alternative, arguments will be presented to describe the ease of use of the different development methods. The most basic case will be described first and the various improvements will follow.

The following criteria are used in the discussion about ease of development:

- **Length of Compile-Execute Cycle** During device driver development, there is a certain time after the driver has been modified before it can be tested. This is because the computer often has to be rebooted between each test run or because a driver that is in the development phase could crash or corrupt the operating system. If the Compile-Execute-Debug cycle could be speeded up, it would reduce the overall development time.
- **Amount of System Knowledge Needed** This criterion is used to gauge the amount of knowledge that is needed about an operating system kernel before a device driver could be implemented.
- **Ease of Debugging** Device drivers are not easy to debug, because they are asynchronous servers and also because they interact at a low level with the hardware of the CPU and hardware of the peripheral devices. It is therefore important that a development method facilitates the ease of debugging.
- **Extensibility** This criteria estimates how easy it is to extend an existing driver to support a related peripheral device.
- **Safety** This measures how much a malfunctioning device driver can influence the rest of the system.

No attempt was made to measure the extent to which each of these criteria are supported by the different development methods. Instead an informal argument is presented to compare the different development methods.

	In-kernel	Extensible	Loadable	User-Level
Long Compile-Execute Cycle	Yes	-	No	No
Extensive System Knowledge Needed	Yes	No	-	No
Easy to Debug	No	-	Yes	Yes
Extensible	No	Yes	-	No
Safe	No	-	-	Yes

Table 1: Ease of development for different development strategies

4.4.1 In-Kernel Device Drivers

System Knowledge Needed

In the most basic case, interrupt handlers for in-kernel device drivers are called directly by the hardware when an interrupt occurs. This scenario allows the device driver developer the most freedom, but it also carries the greatest burden. The developer has the responsibility to save and restore the registers of the interrupted thread, to acknowledge hardware interrupts and if the kernel can be interrupted, to make provision for nested interrupts as well as to enable and disable interrupts. The next process that should execute must also be selected and a context switch must be performed to that process. If the developer of the device driver refrains from performing any of these steps, the operating system could malfunction. This is an unnecessary burden on the device driver developer. Of course, library calls would normally be provided to ease the burden, but the right calls must still be performed in the right order and circumstances. In the Gneiss kernel, all these steps are performed in the generic interrupt handler and the device specific handler need only to perform device specific processing.

The developer must also know how to access buffers in user space and must be able to manage the concurrency between the peripheral device and the processor. This requires that the developer has a good knowledge of the internal working of the kernel and how the kernel interacts with device drivers. After this knowledge has been attained, development becomes easier.

Compile-Execute Cycle

The driver is part of the operating system kernel and the kernel must therefore be rebooted for each test run. This means that the compile-execute cycle of an in-kernel device driver is relatively long. The developer usually needs two computers, one for development and one for testing.

Debugging support

In many cases the only available debugging support for kernel programming consists of writing trace output to the screen or to the serial port. This is at best a time consuming and inefficient mechanism. Newer kernels also contain support for more sophisticated debugging aids like source level debuggers which allow the data structures of a kernel device driver to be examined at run time. However, these debugging aids are only available if the kernel was compiled specifically with debugging support and for the greatest flexibility a remote debugger should be used. The debugging setup then consists of a target machine that executes the kernel with the driver that must be debugged and a host machine that provides the user interface and debugging control, for example to set breakpoints and resume the target machine after it was stopped at a breakpoint. The same amount of flexibility cannot normally be achieved if local debugging is used, since a kernel cannot support a debugging interface while it is being stopped at a breakpoint. However, non-intrusive operations like examining the contents of data structures, can still be performed. Remote debugging also has a number of drawbacks, namely that two computers must be used and it is usually slower than debugging on one machine, due to the overhead of the communication protocol. Furthermore, it is not possible to debug the driver that is used for communication between the host and target machines since this will disrupt the process of communication and therefore the debugging session itself.

Extensibility

A device driver is extensible if a new device driver can be added to an existing framework without the need to examine or modify the framework itself.

Traditional in-kernel drivers are not extensible, since each new device driver must be written from scratch. This increases the development time, even though the usual approach is to copy a driver that is similar and modify only the relevant parts. This means that the developer must understand the working of the original driver and it is easy to introduce errors due to subtle differences in working between the original and the new device.

Safety

In-kernel drivers are unsafe, since a deficient driver could corrupt the kernel and bring the whole system to a halt.

4.4.2 Extensible Device Drivers

Extensible device drivers can be implemented either in the kernel or in user-space. The properties of either user-space or in-kernel device drivers will therefore also be applicable to the relevant extensible device driver.

System Knowledge Needed

An extensible device driver consists of two parts, namely (1) the generic (or framework) part and (2) the specific part. The system knowledge that is needed for the generic part of the device driver is the same as the system knowledge needed for a device driver of any of the other categories. Furthermore, a good knowledge of object oriented design as well as knowledge about the working of a number of similar devices is needed. The developer of the generic driver must also know and understand the kernel support for generic drivers. The combination of these factors means that the development of generic driver along with the development of the first specific driver usually takes longer to complete than a non-extensible driver would have taken. However, the development process is simplified enormously for all other devices drivers that make use of the generic framework. This is because the developer of the specific driver does not need to understand the interaction between the device driver and the kernel, and therefore much of the flow of control of the device is hidden from him.

A generic device driver therefore consists of a tradeoff between the initial development cost and the perceived benefits in terms of extensibility and ease of development later on. In general it is not worthwhile to develop an extensible driver for only one device. There must be at least two or preferably more devices that have enough similarities so that a generic driver can be developed.

Compile-Execute Cycle, Ease of Debugging and Safety

The extensibility of a device driver has little or no impact on the the length of the compile-execute cycle, the ease of debugging, or the safety of the driver.

Extensibility

It is difficult to design and implement a device driver that is so extensible that it does not need to be modified again. This is because new peripheral devices do not necessarily fit

into existing generic frameworks, which forces the framework to be modified again. Another problem occurs when a framework is used to implement a device that does not really fit into that framework. For example, it is difficult to implement a floppy device driver using a generic driver that is aimed at the development of IDE hard disks. This is because even though there are many similarities between a floppy drive and a hard drive, the floppy drive controller is more primitive and therefore the floppy driver must explicitly deal with many aspects of disk operation that are hidden by a hard drive controller.

4.4.3 Loadable Device Drivers

Loadable device drivers is a development method that can be combined with any of the other development methods. It is aimed at reducing the length of the compile-execute cycle and making it easier to configure the system. Therefore only the length of the compile-execute cycle is relevant to this discussion.

The Compile-Execute Cycle

A loadable device driver reduces the length of the compile-execute cycle by allowing device drivers to be loaded and unloaded dynamically. Loadable modules also have other benefits, such as making it easier to configure the system and to reduce system memory usage by allowing seldomly used drivers to be unloaded.

Loadable modules allow a device driver developer to compile and test a driver, without having to reboot the computer. Unfortunately sometimes it is still necessary to reboot, or make use of a second computer. An incorrect device driver could put the peripheral device in a state from which it cannot recover without having to reboot. It is furthermore impractical to use a system to develop a driver that is critical to the working of that system, for example a keyboard driver.

Despite these problems, loadable modules are ideal for the development of drivers for various hardware devices, for example floppy drives, CD-ROMs, scanners, printers and other non-critical devices.

4.4.4 User-Level Device Drivers

User-level device drivers as development method stands in direct conflict with in-kernel drivers. A driver must be implemented as either a user-level driver or an in-kernel driver but it cannot be both.

System Knowledge Needed

This is one of the major advantages of user-level device drivers. The development work is done outside of the kernel and the developer does not need to understand the inner working of the kernel. Communication with the kernel is done via the same system call methodology that any other user program employs. The developer can therefore make use of experience that was gained during the development of such programs. It is not necessary to delve into the mysteries of the kernel and learn a new programming paradigm.

Compile-Execute Cycle

User-level drivers can be loaded and unloaded like any other user-level program. This endows them with the same advantages as loadable modules with respect to the reduction of the *Compile-Execute Cycle*. Of course the same disadvantages also apply, namely it is still necessary to reboot when a device driver that is critical for the execution of the operating system is developed. In a distributed operating system, user-level drivers can also be loaded on a separate test machine for each test run. This is faster than rebooting the test machine all the time.

Debugging support

Since user-level drivers are just specialized user-level programs, all the debugging tools that are available to normal user programs can be used to debug user-level device drivers. This means that breakpoints can be set and data structures and variables can be examined.

Extensibility

User-level drivers are not necessarily extensible, but nothing prevents the implementation of extensible user-level device drivers.

Safety

A user-level driver executes within a protected address space. Therefore it cannot access and corrupt memory that belongs to the kernel or other VMs. On the i386 hardware it is also possible to regulate access to the I/O ports. The kernel can therefore grant access to safe ports and prevent access to critical ports like those that control the operation of the system timer.

Chapter 5

Conclusion

The goal of this thesis was to perform an experiment to compare different strategies to develop device drivers. The comparison was done within the same environment and focused on the ease of development and efficiency issues. As a result of this study, it is recommended that user-level device drivers be chosen as a general device driver development strategy. The ease of development, protection and maintainability of user-level device drivers compensates for the loss of efficiency compared to in-kernel drivers. User-level device drivers also make system configuration easier, since such drivers can be loaded and unloaded like any other user program. However, in certain circumstances in-kernel drivers might be more suitable. For example, drivers for devices that issue a large number of interrupts or drivers that need to be extremely efficient should be implemented within the kernel. Extensible drivers in combination with user-level or in-kernel drivers can also be used with good results if a large number of similar devices must be developed. It is advised that the generic part of the extensible driver be developed by an expert.

The suitability of the event model and the thread model for user-level drivers has also been investigated. It has been argued that in terms of functionality, the event model and the thread model are equivalent. However, in some cases one model is more convenient than the other. The other important factor in terms of device driver development is the type of synchronization primitives that are available, as well as whether the kernel supports synchronous or asynchronous kernel calls.

During this project I have gained much knowledge about system programming, device drivers and the Intel 80386 processor and its successors. I have studied the internal details of a typical micro-kernel (the Gneiss kernel) and its components, namely the scheduler, IPC, thread and

VM creation and device drivers. I have also gained first-hand knowledge of the possible pitfalls that accompany any modification to the kernel.

The development of Event Driven VMs has been curtailed by the need to remain compatible with the Gneiss kernel, which has been designed to support the thread model. Certain compromises, like blocking events and event handlers that execute until completion, could have been eliminated with a different design. Therefore the insights that were gained in device driver development and the study of the thread model and event model should be used to design a new kernel.

This future kernel must allow the thread and event model to be used interchangeably in a natural and efficient way. This will enable the user to choose the paradigm that is most suited to his specific problem. It must also support the development of efficient user-level device drivers. Another possible feature would be language support for cross-address space message passing and event dispatching.

As long as new peripheral devices are being developed, device drivers will have to be written to support these devices. I hope that the information that was presented in this thesis will be of benefit to the field of device driver development.

Bibliography

[1] Eric W. Anderson. The Performance of the Container Shipping I/O System. *Proceedings of the 15th ACM Symposium on Operating System Principles*, 29(5):229–229, 1995.

[2] G. R. Andrews. *Concurrent Programming.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[3] M. Beck et al. *Linux Kernel Internals.* Addison-Wesley, Harlow, England, First edition, 1996.

[4] Thomas Berlage. *OSF/Motif Concepts and Programming.* Addison-Wesley, Wokingham, England, 1991.

[5] Brian N. Bershad et al. Extensibiilty, Safety and Performance in the SPIN Operating System. *Proceedings of the ACM Symposium on Operating System Principles*, 15:267–284, 1995. SPIN.

[6] Brian N. Bershad et al. SPIN - An Extensible Micro-kernel for Application-specific Operating System Services. *ACM Operating Systems Review*, 29(1):74–77, 1995. SPIN.

[7] J. B. Chen and B.N. Bershad. The impact of operating system structure on memory system performance. *Proceedings of the 14th ACM Symposium on Operating System Principles*, 29(4):120–133, December 1993. Ultrix/Mach.

[8] Stephen Coffin. *UNIX System V Release 4 the Complete Reference.* McGraw-Hill, Berkeley, California, First edition, 1991.

[9] M. Condict et al. Microkernel Modularity with Integrated Kernel Performance. Technical report, OSF Research Institute, Cambridge, 1994.

[10] E.C. Cooper and R.P. Draves. C threads. Technical Report CMY-CS-88-154, School of Computer Science, Carnegie-Mellon University, 1988.

[11] J. Crawford and P. Gelsinger. *Programming the 80386.* SYBEX Inc., 2021 Challenger Driver 100, Alameda, California, 1987.

[12] H. M. Deitel. *Operating Systems.* Addison Wesley, Second edition, 1990.

[13] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[14] F. Douglis, M. F. Kaashoek, A. S. Tanenbaum, and J. K. Ousterhout. A Comparison of Two Distributed Systems: Amoeba and Sprite. Report IR–230, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, February 1991.

[15] Alessandro Forin, David Golub, and Brian Bershad. An I/O System for Mach 3.0. *Proceedings of the Second Usenix Mach Symposium*, 1991.

[16] W. Fouché. An Efficient Kernel to Support the Client-Server Model. Master's thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 1991.

[17] W. Fouché and P. de Villiers. A Reusable Kernel for the Development of Control Software. In M. Linck, editor, *6-th Southern African Computer Symposium*, pages 83–94, July 1991.

[18] David Golum et al. Unix as an Application Program. *Proceedings of the USENIX Summer Conference*, June 1990.

[19] M. Guillemont. The Chorus Distributed Operating System: Design and Implementation. In P. C. Ravasio, G. Hopkins, and N. Naffah, editors, *Local Computer Networks*, pages 207–223. Proceedings of the IFIP TC 6 International In-Depth Symposium on Local Computer Networks, Florence, Italy, (19-21 April), North-Holland Publishing Company, 1982.

[20] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, 1992. ISBN 1-880446-42-1.

[21] Intel. *i486 Microprocessor.* Intel Corporation, 1990.

[22] Intel. *Pentium Family User's Manual Volume 3: Architecture and Programming Manual.* Intel Corporation, 1996.

[23] Intel. *Pentium Family User's Manual Volume 3: Architecture and Programming Manual.* Intel Corporation, 1997.

[24] Elias Israel and Erik Fortune. *The X Window System Server.* Digital Press, New York, 1992. ISBN 1-55558-096-3.

[25] Paul A. Karger. Using Registers to Optimize Cross-Domain Call Performance. In *Architectural Support for Programming Languages and Operating Systems*, volume 23, pages 194–204. SIGARCH,SIGPLAN,SIGOPS, April 1989.

[26] Jay Lepreau et al. In-kernel servers on mach 3.0. *Proceedings of the Third Usenix Mach Symposium*, April 1993.

[27] Jochen Liedtke. Improving ipc by kernel design. *Proceedings of the 14th ACM Symposium on Operating System Principles*, 29(4):175–188, December 1993. L3.

[28] Jochen Liedtke. On micro-kernel construction. *Proceedings of the 15th ACM Symposium on Operating System Principles*, 29(5):237–250, 1995. L3/L4.

[29] Jochen Liedtke. Toward real microkernels. *Communication of the ACM*, 39(9):70–77, 1996.

[30] Jochen Liedtke et al. Two years of experience with a micro-kernel based os. *ACM Operating Systems Review*, 25(2):51–62, 1991. Proceedings of the 13th ACM Symposium on Operating Systems Principles.

[31] Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2*, pages 1–200. Springer-Verlag, Berlin, 1993.

[32] P. Muller and P.J.A. de Villiers. Using Oberon to design a hierarchy of extensible device drivers. In P. Schulthess, editor, *Joint Modular Languages Conference*, pages 147–158, Universitätsverlag Ulm GmbH 1994, Postfach 42 04, 89032 Ulm, September 1994.

[33] Adrian Nye. *Xlib Programming Manual.* O'Reilly and Associates Inc., 1989.

[34] M. Reiser and N. Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula.* Addison-Wesley, ACM Press, New York, 1992.

[35] M. D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[36] Andrew S. Tanenbaum. *Operating Systems Design and Implementation.* Prentice Hall, Upper Saddle River, New Jersey 07458, first edition, 1987.

[37] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice Hall, International (UK) Limited, London, Fourth edition, 1999.

[38] Andrew S. Tanenbaum et al. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[39] Andrew S. Tanenbaum and Albert S. Woodhill. *Operating Systems Design and Implementation.* Prentice Hall, Upper Saddle River, New Jersey 07458, Second edition, 1997.

[40] A.S. Tanenbaum, M.F. Kaashoek, R. van Renesse, and H. Bal. The amoeba distributed operating system - a status report. *Computer Communications*, 14(2):324–335, July 1991. Amoeba.

[41] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler.* Addison-Wesley, ACM Press, New York, 1992.